



**POLITECNICO
DI MILANO**

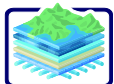
**Università
di Verona**



GeoUML Model

Geometric Model and OCL Constraints Templates

November 2011



SpatialDBgroup

SpatialDBgroup@polimi.it
<http://SpatialDBgroup.polimi.it>

Document title	GeoUML Model – Geometric Model and OCL Constraints Templates
Authors	Giuseppe Pelagatti, Alberto Belussi, Mauro Negri
Version	1.0
Date	15/11/2011

Index

1	INTRODUCTION	5
1.1	OBJECTIVES	5
1.2	BASIC CONCEPTS OF THE GEOUML APPROACH	5
1.3	CONCEPTUAL AND PHYSICAL LEVEL – IMPLEMENTATION MODELS	5
2	GENERAL FEATURES OF THE MODEL	6
2.1	MODEL COMPONENTS	6
2.2	APPROACH FOR THE MODEL DEFINITION AND RELATIONS WITH OTHER ISO STANDARDS	6
2.3	SYNTAX OF THE SPECIFICATION LANGUAGE OF GEOUML	6
3	STRUCTURAL ELEMENTS.....	7
4	GEOMETRIC MODEL	9
4.1	GENERAL FEATURES OF THE GEOMETRIC TYPES AND OBJECTS.....	9
4.2	GEOMETRIC TYPES.....	11
4.2.1	<i>GU_Object</i>	11
4.2.2	<i>GU_Object2D e GU_Object3D</i>	13
4.2.3	<i>GU_PrimitiveObject2D and GU_PrimitiveObject3D</i>	13
4.2.4	<i>GU_Point2D and GU_Point3D (Point)</i>	13
4.2.5	<i>GU_CPCurve2D and GU_CPCurve3D</i>	14
4.2.6	<i>GU_CPSimpleCurve2D and GU_CPSimpleCurve3D (Composite Simple Curve)</i>	16
4.2.7	<i>GU_CPRing2D and GU_CPRing3D (Composite Ring)</i>	16
4.2.8	<i>GU_CPSurface2D</i>	16
4.2.9	<i>Generic aggregate types GU_Aggregate2D and GU_Aggregate3D</i>	18
4.2.10	<i>GU_CXPoint2D e GU_CXPoint3D (Complex Point)</i>	20
4.2.11	<i>GU_CXCurve2D e GU_CXCurve3D (Complex Curve)</i>	20
4.2.12	<i>GU_CXRing2D and GU_CXRing3D (Complex Ring)</i>	22
4.2.13	<i>GU_CNCurve2D and GU_CNCurve3D (Connected Curve)</i>	22
4.2.14	<i>GU_CXSurface2D (Complex Surface)</i>	22
4.2.15	<i>GU_CPSurfaceB3D/GU_CXSurfaceB3D (Composite/Complex Surface Boundary 3D)</i>	24
4.2.16	<i>gUnion (geometric union) and gIntersection (geometric intersection) functions</i>	25
4.3	TOPOLOGICAL RELATIONS	27
5	GEOMETRY-DEPENDENT ATTRIBUTES	30
5.1	INTRODUCTION	30
5.2	SEGMENTED ATTRIBUTE	30
5.3	EVENTS ATTRIBUTE	33
5.4	SUBREGIONS ATTRIBUTE.....	34
6	SPATIAL INTEGRITY CONSTRAINTS.....	37
6.1	INTRODUCTION	37
6.2	TOPOLOGICAL CONSTRAINTS	38
6.2.1	<i>Basic existential topological constraint</i>	39
6.2.2	<i>General rules for constraint formulation</i>	40
6.2.3	<i>Formal definition of the existential constraint using OCL translation rules</i>	42
6.2.4	<i>Variants of the basic existential topological constraint</i>	44
6.2.4.1	<i>Existential topological constraint with selections</i>	44
6.2.4.2	<i>Existential topological constraints on the boundary or planar projection</i>	45
6.2.4.3	<i>Topological constraint linked to an association</i>	46
6.2.4.4	<i>Constraints on segmented or subregions attributes</i>	47
6.2.5	<i>Topological Constraint on union</i>	48
6.2.6	<i>Universal topological constraint</i>	49
6.2.7	<i>Topological constraints with multiple constraining classes</i>	50
6.2.8	<i>Disjunction of topological constraints</i>	51
6.3	COMPOSITION CONSTRAINTS (PART_WHOLE CONSTRAINTS).....	52
6.3.1	<i>Composition constraint</i>	53

GeoUML Model

Geometric Model and OCL Constraints Templates

6.3.2	<i>Constraint of belonging</i>	54
6.3.3	<i>Partition constraint</i>	55
6.3.4	<i>Composition constraints with multiple constraining classes</i>	56
APPENDIX A – TRANSLATION OF CONSTRAINTS IN OCL		57
A.1.	INTRODUCTION.....	57
A.2.	EXISTENTIAL TOPOLOGICAL CONSTRAINT.....	57
A.2.1	<i>Basic form</i>	57
A.2.2	<i>Variant with selection</i>	59
A.2.3	<i>Variant on the boundary or planar projection</i>	62
A.2.4	<i>Variant linked to an association</i>	63
A.3.	UNION TOPOLOGICAL CONSTRAINT.....	64
A.3.1	<i>Basic form</i>	64
A.3.2	<i>Variant with selection</i>	64
A.3.3	<i>Variant with selection and segmented attributes</i>	65
A.4.	UNIVERSAL TOPOLOGICAL CONSTRAINT.....	67
A.4.1	<i>Basic form</i>	67
A.4.2	<i>Variant with selection</i>	67
A.4.3	<i>Variant with selection and segmented attributes</i>	68
A.5.	COMPOSITION CONSTRAINT.....	70
A.5.1	<i>Basic form</i>	70
A.5.2	<i>Variant with selection</i>	70
A.5.3	<i>Variant with selection and segmented attributes</i>	71
A.5.4	<i>Variant on boundary and planar projection</i>	72
A.5.5	<i>Variant linked to an association</i>	73
A.6.	CONSTRAINT OF BELONGING.....	74
A.6.1	<i>Basic form</i>	74
A.6.2	<i>Variant with selection</i>	75
A.6.3	<i>Variant with selection and segmented attributes</i>	76
A.6.4	<i>Variant on boundary and planar projection</i>	78
A.7.	PARTITION CONSTRAINT.....	79
A.8.	COMPOSITION CONSTRAINTS WITH MULTIPLE CONSTRAINING CLASSES.....	80
A.8.1	<i>Basic form</i>	80
A.8.2	<i>Variant with selection</i>	81
A.8.3	<i>Variant with selection and segmented attributes</i>	82
A.8.4	<i>Variant on boundary and planar projection</i>	85

1 Introduction

1.1 Objectives

This document defines the main components of the *GeoUML model* and in particular it describes in details: (i) the set of geometric types defined for the specification of the spatial components of the classes (spatial attributes) and (ii) the set of OCL Templates that have been defined for the specification of spatial integrity constraints among the spatial attributes of the classes.

The document is organized as follows: in Section 2 the general features of the model are presented; Section 3 illustrates the structural elements of the model; in Section 4 the geometric model is formally defined, while in Section 5 and 6 the segmented and subregion attributes and the OCL templates for the specification of integrity constraints are illustrated.

1.2 Basic concepts of the GeoUML approach

The GeoUML model has been designed for the definition of the structural part, called *Conceptual Schema*, of a *Content Specification* regarding geographical data, that usually includes also textual descriptions concerning survey details. It was the result of an Italian project that aimed to define the Italian National Core as a reference content specification for the creation of topographical databases at regional and local (municipality) level.

A *Content Specification* is a formal definition of the data that must be contained in a *Data Product* when it is created by a specific organization. The term *Data Product* refers to the definition that is presented in the ISO 19100 standards and indicates an organized and consistent collection of geographical information. A *Data Product* could be for example a set of files or a database. Given a *Conceptual Schema* there might possibly exist several *Data Product* implementing that schema.

1.3 Conceptual and physical level – Implementation models

A *Conceptual Schema* defines the properties that a *Data Product* must have at conceptual level, that is independently from the technology that has been chosen to implement it.

Given a *Conceptual Schema CS* we can define a set of rules that allows one to implement in a specific technology a *Data Product* that represents the content, which is described by *CS*. More specifically the application of such rules allows one to produce automatically starting from *CS* the corresponding *Physical Schema* that defines the physical data structure of the *Data Product* in a specific technology (shapefiles, SQL database, GML, etc...). This set of rules is called *Implementation Model*.

The main motivation for separating the *Conceptual Schema* from the *Implementation Model* is the possibility to define different *Implementation Models* and to use them for generating several physical schemas starting from the same *Conceptual Schema*.

So the same conceptual description keeps its validity and significance even if a change of technology occurs.

2 General features of the model

2.1 Model components

GeoUML model is composed of a set of *Constructs* that allow one to define formally the conceptual schema of a content specification. Constructs are of two categories:

1. the ***Structural Elements***: that are the constructs to be used for defining the data structures used for the content representation.
2. the ***Integrity Constraints***: that are applied to the structural elements of a schema for defining the properties that must be satisfied by data of any schema consistent Data Product.

2.2 Approach for the model definition and relations with other ISO standards

GeoUML is a specialization of the ISO standards 19103, 19107, 19109 and these standards refer to the standards *UML VI.3* (Unified Modeling Language) e *OCL* (Object Constraint Language). Therefore, the formalization of GeoUML model is obtained by providing the mapping rules that given a GeoUML schema can produce the corresponding UML schema, that is conformant with the above cited standards.

However, for sake of readability and abstraction, we also describe some parts of the model by using an independent definition of the constructs, in particular we chose to follow this approach for the geometric types, since we need to extend the Simple Feature Model to deal with 3D for points and curves and we wanted to avoid the specialization of the Spatial Schema (ISO 19107), which is a huge and advanced standards with respect to the current GIS technology.

Regarding the use of OCL notice that: (i) the function *oclisKindOf()* is renamed to *isKindOf()* and we remind the notation *O.f.g*, where *O* is an object and *f* and *g* are functions that produce set of values, returns a set of values and not a set of sets of values.

2.3 Syntax of the specification language of GeoUML

The reference syntax of GeoUML has a textual form, since this was a strict requirement of the project for which the model is born: the definition of the Italian National Core. However also a graphical syntax is admitted, which is an extension (for representing graphically also integrity constraints) of the well-known UML class diagram syntax.

In particular, in order to improve readability of this document, the language keywords are in italic and underlined (e.g. *class*).

3 Structural elements

The basic structural elements of GeoUML model are the following constructs:

- class
- attribute (non geometric)
- cardinality
- enumeration
- hierarchical enumerations
- association
- inheritance
- geometric attribute
- attribute of geometric attribute
- primary key
- topological layer

All basic structural elements have the following properties:

- **Name** (compulsory): is the word (or set of words) that identifies the concept in the application domain that is represented by the element in the conceptual schema.
- **Code** (compulsory but not for constraints): is an alphanumeric code that identifies the element in the schema.
- **Alphanumeric code** (compulsory only for classes): it is a short form of the name.

All the basic elements of GeoUML are formally defined by given the corresponding mapping rule towards the cited ISO standards. In particular, the rules of ISO 19109, “*Rules for Application Schema*” [19109] have been applied.

For example we report hereby the mapping rule for the class construct with some alphanumeric attributes, which is straightforward.

Mapping to ISO Application schema

The following table presents the mapping between GeoUML basic alphanumeric types and UML-ISO types.

GeoUML basic alphanumeric types	UML-ISO types
<i>Integer</i>	Integer
<i>Real</i>	Double
<i>String(N)</i>	CharacterString
<i>NumericString(N)</i>	CharacterString with a restriction only to the set of digits.
<i>Boolean</i>	Boolean
<i>Time</i>	Time
<i>Date</i>	Date
<i>DateTime</i>	DateTime

GeoUML Model

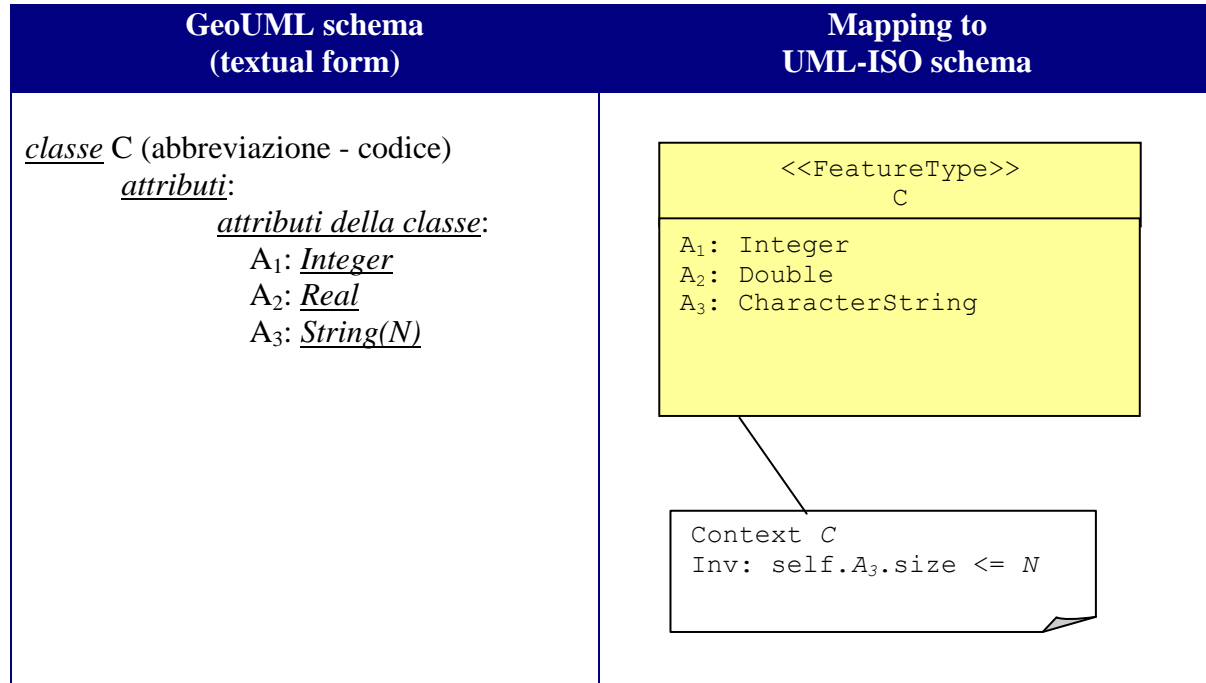
Geometric Model and OCL Constraints Templates

Mapping rule for Classes with alphanumeric attributes only.

Basic Class Rule

Given a class *C* of a GeoUML schema a corresponding class with the stereotype `FeatureType` and with the same name is generated in the UML-ISO schema. For each alphanumeric attribute of *C* a corresponding UML-ISO attribute is generated and its type is chosen according to the above mapping table.

Example:



The complete list of the mapping rules is reported in the SpatialDBGroup documentation published at (in Italian):
<http://spatialdbgroup.polimi.it/fileadmin/docs/it/GeoUMLCISISrevisioneMAGGIO2010.pdf>

4 Geometric model

4.1 General features of the geometric types and objects

The geometric model defines a set of types that can be used as domains for the geometric attributes of a GeoUML class.

The geometric types allow one to define two categories of geometric objects:

- **Geometric primitives:** atomic geometries (without any subdivisions in parts) which are composed of a single connected and homogeneous element of the reference space (e.g. a curve)
- **Geometry collections:** set of geometric primitives that can be homogeneous in the component types (multi-points, multi-curves or multi-surfaces) or heterogeneous (geometry collection); in some cases it is required the satisfaction of some spatial integrity constraints by collection components.

GeoUML geometric types are defined as UML classes and are organized in a UML class hierarchy shown in Figure 4.1. This approach allows one to describe incrementally the properties of the types, starting with the common properties which are illustrated in the root class of the hierarchy called *GU_Object* (which is an abstract class – abstract types cannot be used as attribute domain).

All geometric objects in GeoUML are defined in a precise coordinate reference system. According to this feature GeoUML types are classified in two categories:

- types that describe geometric objects without the third coordinate (Z), called 2D types. The type *GU_Object2D* is the root of the sub-tree in the type hierarchy of Figure 4.1.
- types that describe geometric objects in the 3D space, called 3D types. The type *GU_Object3D* is the root of the sub-tree in the type hierarchy of Figure 4.1.

Notice that Figure 4.1 highlights also the associations among aggregate types (geometry collection) and component types (geometric primitives).

This section defines for each type the properties of the admitted geometries by referring to the abstract concept of point sets.

Notice that the specific representation of the geometries at physical level, including the interpolation methods, is not considered by the GeoUML geometric model. Indeed, they will be specified in the Implementation Model that will be chosen for the Data Product implementing a GeoUML schema. For example the type *GU_CPCurve* at conceptual level defines the properties of a curve in the Euclidian space and can be used as domain of an attribute of a GeoUML class, but its physical representation in an implementation model based on the Simple Feature Model could be a Linestring type, that represents curve geometries as the concatenation of segments with linear interpolation.

In section 4.2 we illustrate in details the geometric types, while in section 4.3 the operators used for testing the topological relations among geometric types are separately presented.

GeoUML Model

Geometric Model and OCL Constraints Templates

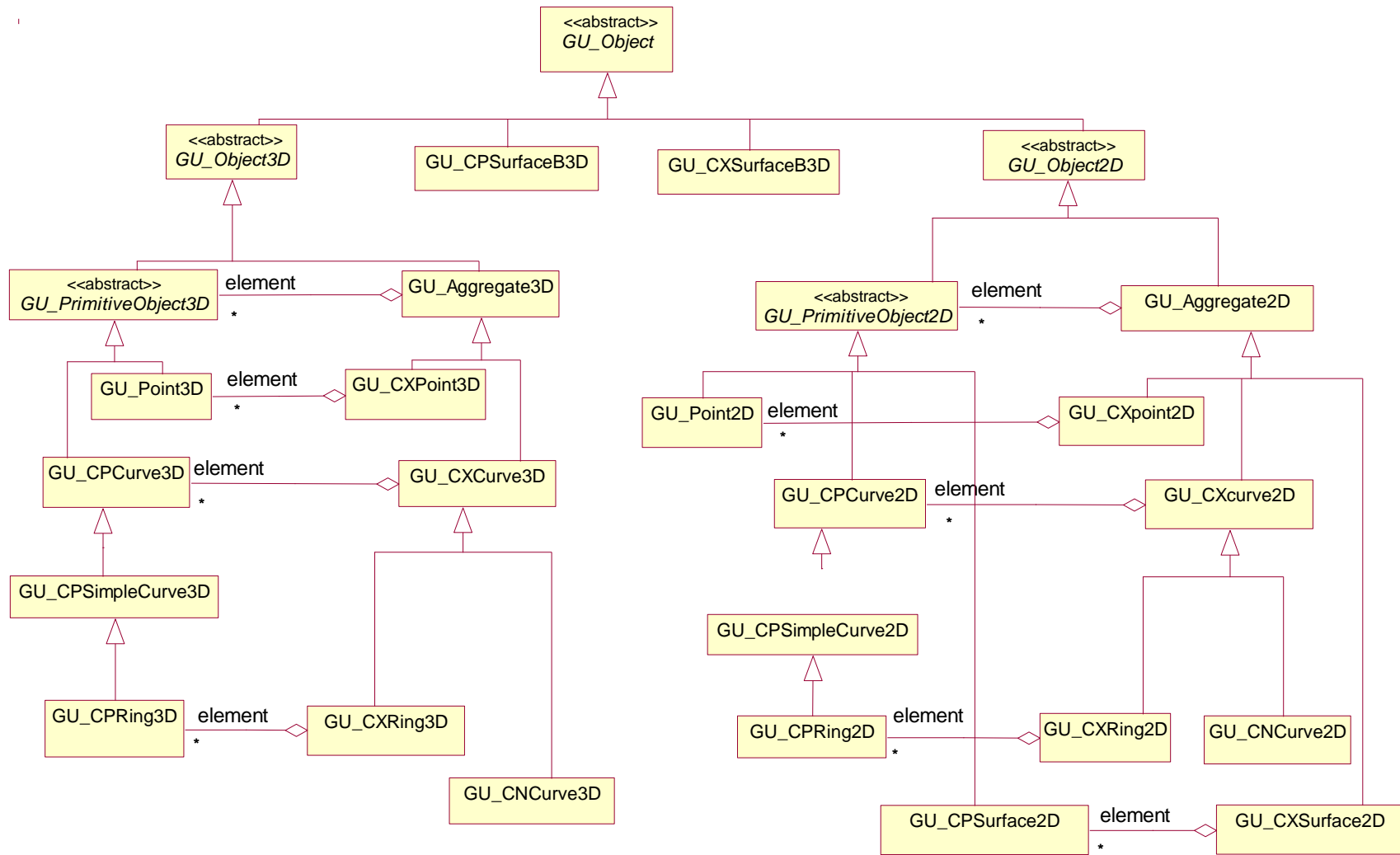


Figure 4.1 UML hierarchy of the classes representing the available geometric types of the GeoUML.

The GeoUML geometric model enriches the standard SFM mainly in the following items:

- The extension to the 3D space of the geometries for representing points and curves together with the operators that test topological relations (the test is performed in 3D space)
- The extension to the 3D space of the boundary of the 2D surfaces by introducing the type “surface with 3D boundary”
- The introduction of some specializations of the geometric types for representing curves.

The name adopted for GeoUML geometric types have been chosen in the context of the project “Italian National Core”.

In the sequel, for sake of simplicity, when it is necessary to refer to a subset of geometric types we use the symbol “*”; for example, *C*curve*D* means *CPcurve2D* or *CPcurve3D* or *CXcurve2D* or *CXcurve3D*.

4.2 Geometric types

In this section we define in details the geometric types of GeoUML by describing the semantics of the general properties and specific properties. Moreover, for each type the point set definition of the admitted geometries (domain values) is presented.

4.2.1 GU_Object

Definition of the domain values

GU_Object is an abstract type and contains the definition of general properties that are shared by all geometric objects of the GeoUML geometric model.

From a mathematical point of view an object of type *GU_Object* is a infinite point set (except for the types that represent isolated points) with the following properties:

1. It is defined in a Euclidean space \mathfrak{R}^n where the n coordinates of a point are assigned on the basis of the adopted reference coordinate system. In GeoUML the space \mathfrak{R}^2 is defined for objects in the 2D space and \mathfrak{R}^3 is also defined for objects in the 3D space, where the Z coordinate is used for the representation of the height above sea level (altitude);
2. It is **topologically closed**, which means that the point set representing the object includes also the points the describe the boundary of the set.
3. It is **regular**, which means that, given the point set representing the object the union of its interior and its boundary is equal to the set itself; this property avoids to represent tricky objects like for example polygons with cuttings in their interiors or polygon holes composed of a single point, etc...

General Properties.

- **boundary():** *GU_Object*

It returns the boundary of the geometric object that is defined by the point set that limits the extension of the object, that is by the points that satisfy the following property: given any neighborhood of them, it intersects both their interior and exterior. The calculation of the boundary is performed by supposing that the embedding space has the same dimension of the object (e.g. a 2D space for surfaces and a 1D space for curves); this rule permits to obtain correct boundaries with respect to the user expectation (e.g. the boundary of a curve will be its endpoints; while by embedding the curve in a 2D space, the whole curve will become boundary of itself. As a consequence, the boundary of an object of dimension d will always have a dimension that is $(d-1)$. The detailed description of the geometry representing

the object boundary depends on the specific geometric type and thus will be described in the subclasses of *GU_Object*.

- **coordinateDimension():** Integer
It returns the dimension of the object coordinates, that is the number of the necessary axes for specifying the position of each point of the geometric object in a reference coordinate system. This value depends in GeoUML only on the geometric type to which the object belongs.
- **dimension():** Integer
It returns the inherent dimension of the point set that is associated to the geometric object; for example, a curve has dimension 1, even if it is embedded in a 3D space. It is always equal or less that the coordinate dimension of the same object.
- **isCycle():** Boolean
It returns TRUE if the geometric object is a cycle (this term is often substituted by “is closed”, when it cannot be confused with “is topologically closed”). A cyclic geometric object has an empty boundary.
- **isSimple():** Boolean
It returns TRUE is the geometric object is simple, that is if it does not present self intersection or self tangent points.
- **spatialReferenceSystem():** Integer
It returns the international code that identifies the reference coordinate system of the geometric object.
- **planar():** *GU_Object2D*
It returns a geometric object embedded in the 2D space that describes the point set that is obtained by 2D projecting the point set representing the geometric object. The geometric type of the returned object depends on the input object type and is specified in the subclasses of *GU_Object*.

Point Set Operations: the point set operations (union, intersection and difference) can be defined very easily in *GU_Object*, however, we prefer to present them after the subclasses since the resulting object type can be very complex. For example, the union of two curves is not necessarily a curve, but it is in particular cases.

Function PS(). Given a geometric object O, O.PS() returns the point set associated to O; this function is introduced in order to simplifying the presentation of the definitions.

Comment

Notice that many other properties could have been defined in *GU_Object*. However, we introduce only those ones that are strictly necessary for the definition of spatial integrity constraints on a GeoUML schema.

4.2.2 GU_Object2D e GU_Object3D

Definition of possible values

The abstract types *GU_Object2D* e *GU_Object3D* have been introduced to clearly distinguish types which describe objects represented by point sets in 2D space from those in 3D space.

Specialization of inherited properties

- **coordinateDimension()**

`self.isKindOf(GU_Object2D) ⇒ self.coordinateDimension()=2`

`self.isKindOf(GU_Object3D) ⇒ self.coordinateDimension()=3`

4.2.3 GU_PrimitiveObject2D and GU_PrimitiveObject3D

Definition of possible values

These abstract types represent a generic geometric primitive in 2D and 3D space, respectively, and have been introduced to simplify the definition of generic aggregates.

4.2.4 GU_Point2D and GU_Point3D (Point)

Definition of possible values

A geometric object of the types *GU_Point2D* and *GU_Point3D* is a zero-dimensional object called a “point” which represents a position in a space of 2D and 3D coordinates, respectively.

Specialization of inherited properties

- **boundary()**

`self.boundary() = ∅`

- **dimension()**

`self.dimension() = 0`

- **isCycle()**

`self.isCycle() = true`

- **isSimple()**

`self.isSimple() = true`

- **planar()**

`self.isKindOf(GU_Point2D) ⇒ self.planar() = self`

`self.isKindOf(GU_Point3D) ⇒ self.planar() = q,`

where *q* has the following properties: `q.isKindOf(GU_Point2D)=true` and *q* is obtained from the original object by eliminating the coordinate Z from the original object.

4.2.5 GU_CPCurve2D and GU_CPCurve3D

Definition of possible values

The types *GU_CPCurve2D* and *GU_CPCurve3D* are used to define a one-dimensional object which corresponds to the intuitive concept of a continuous elementary curve obtained by “moving” a point continuously in space, without bifurcations nor break points of continuity. Moreover, no self-intersections on infinite point set are admitted. Examples of correct elementary curves are given in Figure 4.2, while Figure 4.3 shows incorrect examples.

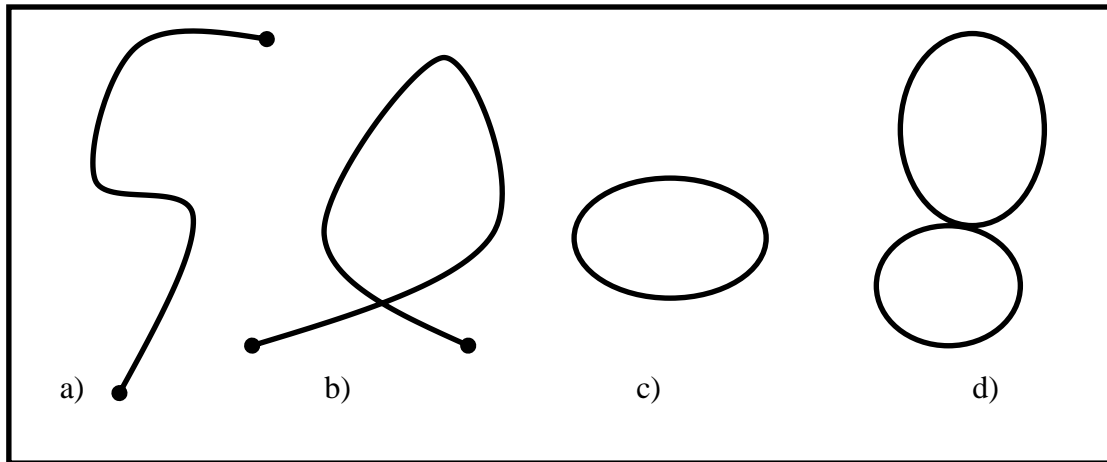


Figure 4.2 - Examples of elementary curves (GU_CPCurve2D).

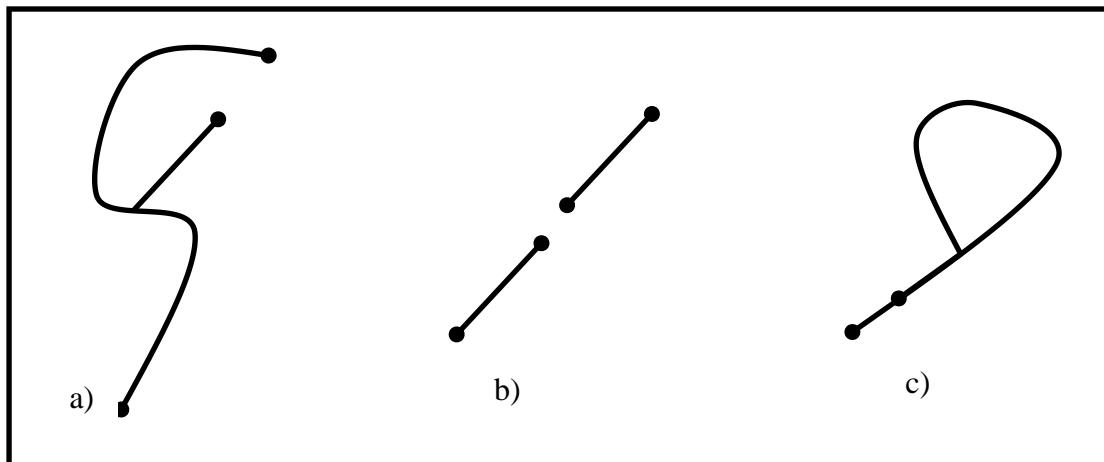


Figure 4.3 – Examples of geometries not describable as elementary curves.

In mathematical terms, given a closed interval of real numbers associated to an object:

$$\text{self.Domain} = [a, b] = \{t \in \mathbb{R} \mid a \leq t \leq b\}, \text{ con } a < b$$

a curve is defined as a point set obtained through a continuous function f

$$\text{self.f} : [a, b] \rightarrow \mathbb{R}^n,$$

where $n=2$ for curves of type *GU_CPCurve2D* and $n=3$ for curves of type *GU_CPCurve3D*.

The curve allows for a maximum of one discrete number of values from the domain for which the function f returns the same point of the considered space.

$\forall x_1, x_2, x_3, x_4 \in \text{self.Domain}$

$((x_1 < x_2 \wedge x_2 < x_3 \wedge x_3 < x_4) \Rightarrow (f([x_1, x_2]) \neq f([x_3, x_4])))$

where $f([x, y])$ indicates the section of curve obtained by applying f to the interval $[x, y]$.

Specialization of inherited properties

- **boundary()**

The boundary of a curve is determined supposing that the curve is defined in a one-dimensional space and thus it is composed of the end points, for open curves (Figure 4.2, cases (a) and (b)), while the boundary does not exist if the curve is closed (Figure 4.2, cases (c) and (d)).

$(\text{self.f}(a) = \text{self.f}(b)) \Rightarrow \text{self.boundary}() = \emptyset$
 $(\text{self.f}(a) \neq \text{self.f}(b)) \Rightarrow \text{self.boundary}() = \{\text{self.f}(a), \text{self.f}(b)\}$

- **dimension()**

$\text{self.dimension} = 1$

- **isCycle()**

for determining whether the curve is closed (Figure 2.2, cases (c) and (d)).

$(\text{self.f}(a) = \text{self.f}(b)) \Rightarrow \text{self.isCycle}() = \text{true}$
 $(\text{self.f}(a) \neq \text{self.f}(b)) \Rightarrow \text{self.isCycle}() = \text{false}$

- **isSimple()**

Indicates true when the curve does not pass twice through the same point (Figure 4.2, case (a)) or the point coincides only with the end point of the curve (Figure 4.2, case (c)). Notice that a non-simple curve may intersect itself only in a discrete number of points (Figure 4.2, cases (b) and (d)),

$\text{self.isSimple}() = \text{true} \Leftrightarrow \forall x_1, x_2 \in \text{self.Domain} ((\text{self.f}(x_1) = \text{self.f}(x_2) \wedge x_1 \neq x_2) \Rightarrow (x_1 = a \wedge x_2 = b))$

- **planar()**

Generally, the projection in 2D space of a 3D primitive curve generates a primitive curve of the same type as the original curve, although in some cases the projected curve generates an object of a different type, such as: a vertical curve made up of vertices in which only the coordinate Z changes can generate in the projection a single point on the plane, a ring on the XZ plane generates a simple curve on the plane XY, and finally a simple curve with segments that overlap or intersect one another in the projection generates an aggregate of curves.

$\text{self.isKindOf}(\text{GU_CPCurve2D}) \Rightarrow \text{self.planar}() = \text{self}$

$\text{self.isKindOf}(\text{GU_CPCurve3D}) \Rightarrow \text{self.planar}() = q,$

where q has the following properties: $q.isKindOf(\text{GU_Object2D}) = \text{true}$ and q is the object that describes the point set obtained by eliminating the coordinate Z from all points of the point set which describe the object self .

4.2.6 GU_CPSimpleCurve2D and GU_CPSimpleCurve3D (Composite Simple Curve)

Definition of possible values

The types *GU_CPSimpleCurve2D* and *GU_CPSimpleCurve3D* are used to define a simple and open curve (Figure 4.2, case (a)).

Specialization of inherited properties

- **isSimple()**
 `self.isSimple() = true`

4.2.7 GU_CPRing2D and GU_CPRing3D (Composite Ring)

Definition of possible values

The types *GU_CPRing2D* e *GU_CPRing3D* are used to define a simple closed curve, corresponding to the intuitive concept of ring (Figure 4.2, case (c)).

Specialization of inherited properties

- **boundary()**
 `self.boundary() = \emptyset`
- **isCycle()**
 `self.isCycle() = true`
- **isSimple()**
 `self.isSimple() = true`

4.2.8 GU_CPSurface2D

Definition of possible values

A geometric object defined by this type is an elementary two-dimensional surface defined in 2D space. An elementary surface is defined by a set of *GU_CPRing2D* rings: a ring, called f_e , represents the external boundary of the surface and a set of zero or more rings, called $F_i = \{f_{i_1}, \dots, f_{i_n}\}$, which represent the internal boundaries that delimit any holes in the surface; since a ring does not intersect itself, a boundary cannot possess loops which would break the connection (defined below) and the surface cannot degenerate to an open curve (this occurs when the external boundary is composed of a single segment moving in one direction and in the reverse direction).

CONTINUA....

The mathematical definition of elementary surface is based on the property of a ring f of dividing the 2D space in two regions (Jordan curve theorem): a closed internal region of a finite area indicated by $\text{Int}(f)$ and an external region of an infinite area indicated by $\text{Ext}(f)$; both regions include the ring f .

A surface S described by the external ring fe and by the set of internal rings Fi is made up of the set of points in 2D space which satisfy the following properties:

1. The surface S is composed of the points that belongs to the internal region defined by the external boundary and to the external regions defined by the internal boundaries, moreover it includes the boundaries (both internal and external) for guaranteeing topological surface closure:

$$S = \text{Int}(fe) \cap \text{Ext}(fi_1) \cap \dots \cap \text{Ext}(fi_n), \text{ con } fi_k \in Fi, \forall k \in [1, n].$$

2. All holes must be contained in the internal region defined by the external boundary and each internal boundary may only touch the external boundary at one point; a hole which touches the external boundary at two points disconnects the surface, or rather the convergence of the external boundary with an internal boundary causes the degeneration of the surface to a curve:

$$\forall fi_k \in Fi \quad (\text{Int}(fi_k) \subset \text{Int}(fe) \wedge ((fi_k.PS() \cap fe.PS() = \emptyset) \vee (|fi_k.PS() \cap fe.PS()|=1))).$$

3. A hole cannot be contained in another hole or overlap it. Moreover, two holes may only touch each other at one point, like in the previous case.

$$\forall fi_k, fi_j \in Fi, (fi_k \neq fi_j \Rightarrow (\text{Int}(fi_k) \subset \text{Ext}(fi_j) \wedge ((fi_k.PS() \cap fi_j.PS()) = \emptyset) \vee (|fi_k.PS() \cap fi_j.PS()|=1))).$$

4. The internal part of the surface S must be connected, such that any two points of the surface S (excluding the boundaries) are connected by a curve that does not cross the boundaries. Formally, given:

- C as the set of all elementary curves of type *GU_CPCurve2D* definable in 2D space

- IS (internal part of S) = $S - (fe.PS() \cup fi_1.PS() \cup \dots \cup fi_n.PS())$

with $fi_k \in Fi, \forall k \in [1, n]$

you have that:

$$\forall p_i, p_j \in IS \quad (p_i \neq p_j \Rightarrow (\exists c \in C \quad ((c.PS() \subset IS) \wedge (c.f(a)=p_i) \wedge (c.f(b)=p_j))))$$

Figures 4.4 e 4.5 show examples of correct and incorrect surfaces, respectively.

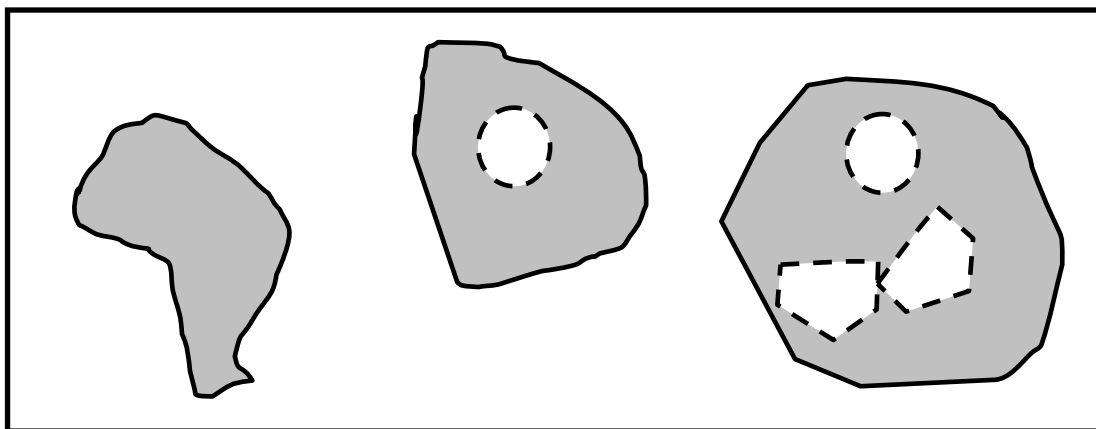


Figure 4.4 – Surface examples (*GU_CPSurface2D*). Dotted lines represent internal boundaries.

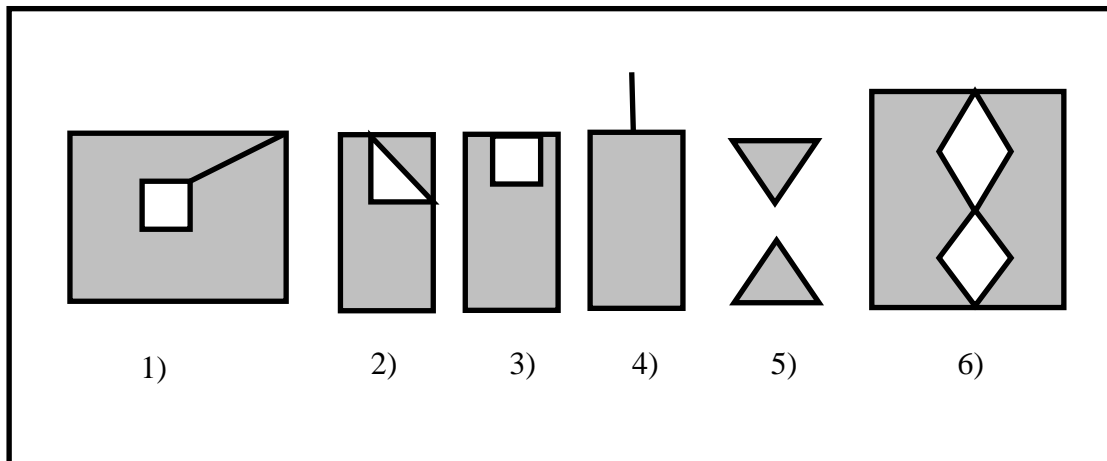


Figure 4.5 – Examples of geometries not describable as objects of type *GU_CPSurface2D*.

Notice that polygons 2, 5, and 6 in Figure 4.5 are describable as aggregates of two surfaces, while the others require the removal of a linear segment to be considered supported surfaces.

Specialization of inherited properties

- **boundary()**
returns an aggregate of type *GU_CXRing2D*, the components of which are rings that represent the external and internal boundaries of the surface.

```
self.boundary().element = {fe, fi1, ..., fik, ..., fin}
con fik ∈ Fi, ∀k ∈ [1, n]
```
- **dimension()**

```
self.dimension() = 2
```
- **isCycle()**
a surface cannot be closed in 2D space.

```
self.isCycle() = false
```
- **isSimple()**
a surface cannot intersect itself in 2D space.

```
self.isSimple() = true
```

4.2.9 Generic aggregate types *GU_Aggregate2D* and *GU_Aggregate3D*

Definition of possible values

Types *GU_Aggregate2D* and *GU_Aggregate3D* are used to define an aggregate, in 2D and 3D spaces, respectively, made up of a collection of zero or more primitive geometric objects (which may be of different types) sharing the same reference system as the aggregate. Aggregates of aggregates are not supported. Finally, a generic aggregate does not place constraints on component geometries (these may also be overlapped and coincide).

From a mathematical perspective, an aggregate *A* is interpreted as the point set obtained from the union of point sets of individual component objects:

$$A.PS() = g_1.PS() \cup \dots \cup g_n.PS(), \quad \forall g_i \in A.element$$

In GeoUML, subtypes of the generic aggregate are defined in order to restrict component types according to dimension: only point in types *GU_CXPoint2D* and *GU_CXPoint3D*, only curves in types *GU_CXCurve2D*, *GU_CXCurve3D*, *GU_CXRing2D*, *GU_CXRing3D*, *GU_CNCCurve2D* e *GU_CNCCurve3D* and only surfaces in type *GU_CXSurface2D*. Finally, in

some types, constraints have been placed on topological relations allowed among aggregate components.

Specialization of inherited properties

- **boundary()**
self.boundary() = null
- **dimension()**
the generic aggregate may contain objects of different dimensions, therefore it is not possible to associate dimension statically with type, as is the case with its subtypes; as such, the dimension of the aggregate is determined by the largest dimension object.
self.dimension() = max({g.dimension() | g ∈ self.element})
- **Dimension of the coordinates of an object**
self.isKindOf(GU_Aggregate2D) ⇒ (self.coordinateDimension() = 2
∧ ∀ g ∈ self.element (g.coordinateDimension() = 2))
self.isKindOf(GU_Aggregate3D) ⇒ (self.coordinateDimension() = 3
∧ ∀ g ∈ self.element (g.coordinateDimension() = 3))
- **isSimple()**
self.isSimple() = null
- **isCycle()**
self.isCycle = null
- **planar()**
returns an object of type *GU_Aggregate2D*
self.planar().element = {g.planar() | g ∈ self.element}

4.2.10 GU_CXPoint2D e GU_CXPoint3D (Complex Point)

Definition of possible values

A geometric object of types *GU_CXPoint2D* and *GU_CXPoint3D* is an aggregate of zero or more points all belonging to types *GU_Point2D* and *GU_Point3D* respectively.

Specialization of inherited properties

- **boundary()**
self.boundary() = \emptyset
- **dimension()**
self.dimension() = 0
- **isCycle()**
self.isCycle() = true
- **isSimple()**
a point aggregate is simple when all points are geometrically disjoint.
self.isSimple() = true
 $\Leftrightarrow \neg \exists g_i, g_j \in \text{self.element} (g_i \neq g_j \wedge (g_i.PS() = g_j.PS()))$

4.2.11 GU_CXCurve2D e GU_CXCurve3D (Complex Curve)

Definition of possible values

An object of types *GU_CXCurve2D* and *GU_CXCurve3D* is a one-dimensional object consisting of a collection of zero or more curves of types *GU_CPCurve2D* and *GU_CPCurve3D*, respectively, which must overlap neither partially nor fully (duplication) in order to keep the property of aggregate boundary constant.

This type is used to define complex curves that support bifurcations and break points of continuity, generating complex curves that may or may not be connected.

Defining the internal part of a curve $c \in \text{GU_CPCurve2D}$ (*GU_CPCurve3D*) as:

$$I(c) = c.PS() - c.boundary().PS()$$

$$\forall c_i, c_j \in \text{self.element} (c_i \neq c_j \Rightarrow ((I(c_i) \cap I(c_j) = \emptyset) \vee (|(I(c_i) \cap I(c_j))| < \infty)))$$

Specialization of inherited properties

- **boundary()**
the boundary of a complex curve contains points of the curve which belong to the boundary of an even number of aggregate component curves (“*mod 2 union rule*” from standard ISO 19125). Let P be the set of all points of type *GU_Point2D* (*GU_Point3D*) of 2D (3D) space:

$$\text{self.boundary()} = \{p \in P \mid \exists g \in \text{self.element.boundary()} (g.PS() = p.PS() \wedge g.isOddBoundary(\text{self.element}))\}$$

where: *g.isOddBoundary(A)* returns true if the point *g* is the boundary of an uneven number of curves of set *A*.

The boundary of the curve in Figure 4.6 a) is made up of 4 outer points, even where the aggregate is composed of 4 simple curves converging at the point of intersection, while the boundary of the curve in Figure 4.5(b) is made up of 3 end points and the internal intersection point, even where the internal point is a boundary of a single curve or of three curves. Like in the case of Figure 4.5(a), the boundary of the curve in Figure 4.5(c) is made up of only the end points of the component curves, while in case (d) the boundary is empty since all components are cycle.

- **dimension()**
self.dimension() = 1
- **isCycle()**
self.isCycle() = true $\Leftrightarrow \forall g \in \text{self.element } (g.isCycle())$
- **isSimple()**
The aggregate is simple when each component curve is simple and the curves only touch one another at the boundary points; this constraint prevents the internal part of two component curves from overlapping, and prevents the boundary point of a curve from touching the internal part of another component curve.
self.isSimple() = true \Leftrightarrow
 $\forall g \in \text{self.element } (g.isSimple()$
 $\wedge (\forall g_i, g_j \in \text{self.element } (g_i \neq g_j \Rightarrow$
 $((g_i.PS() \cap g_j.PS())$
 $=$
 $(g_i.boundary().PS() \cap g_j.boundary().PS()))))$

Figure 4.6 shows a simple aggregate (case (c)), non-simple aggregates (cases (a) and (d)) with simple components; the aggregate in Figure 4.6(b) is simple where it is made up of 3 curves which touch one another at the internal point, while it is not considered simple where it contains 2 curves with one which touches the internal part of another with its own boundary.

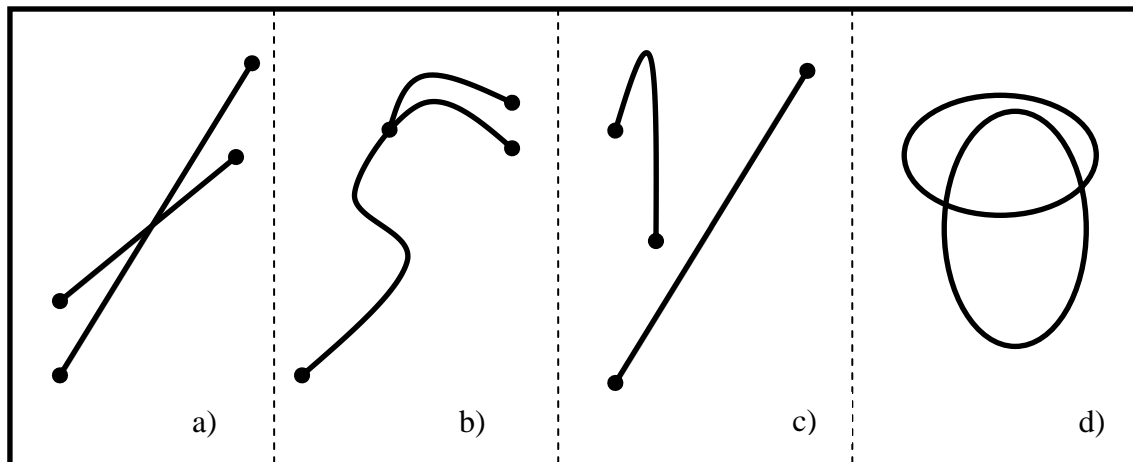


Figure 4.6 – Examples of curve aggregates (*GU_CXurve2D*).

Comment

Notice that there is NO biunique correspondence between a curve viewed as a collection of primitive geometric objects and the point set by which it is represented in space as the point set may correspond to different aggregates of objects; for example, the aggregate in Figure 4.6(b) may be composed of 2, 3, or more primitive curves. The boundary definition of the complex curve is based on the point set described by the aggregate and is thus invariable in respect to the various aggregate objects compositions which correspond to the same point set of the considered space.

4.2.12 GU_CXRing2D and GU_CXRing3D (Complex Ring)

Definition of possible values

Types *GU_CXRing2D* and *GU_CXRing3D* are a specialization of types *GU_CXCurve2D* and *GU_CXCurve3D* respectively. They are used to define a one-dimensional aggregate made up of a collection of zero or more rings of types *GU_CPRing2D* and *GU_CPRing3D* respectively. The restriction on partial or total overlapping of the type *GU_CXCurve* of the corresponding dimension is inherited, though this does not apply any further constraint to the possible topological relations between components.

Specialization of inherited properties

- **boundary()**
self.boundary() = \emptyset
- **isCycle()**
self.isCycle() = true

4.2.13 GU_CNCCurve2D and GU_CNCCurve3D (Connected Curve)

Definition of possible values

Types *GU_CNCCurve2D* and *GU_CNCCurve3D* are specializations of types *GU_CXCurve2D* and *GU_CXCurve3D* respectively, which give the complex curve the property of connection of internal parts: any two points of the complex curve are connected by an elementary curve contained in the complex curve.

Let C be the set of all elementary curves (*GU_CPCurve2D* / *GU_CPCurve3D*)

$$\forall p_i, p_j \in \text{self.PS}() \quad (p_i \neq p_j \Rightarrow (\exists c \in C \quad (c.PS() \subset \text{self.PS}() \wedge c.f(a) = p_i \wedge c.f(b) = p_j)))$$

4.2.14 GU_CXSurface2D (Complex Surface)

Definition of possible values

An object of type *GU_CXSurface2D* is a complex surface comprising a collection of zero or more surfaces of type *GU_CPSurface2D* that are disjoint or may only touch themselves through points of the boundary (therefore the complex surface is generally a non-connected objects):

Let $I(g)$ be the internal part and $F(g)$ be the boundary of a surface $g \in \text{GU_CPSurface2D}$ defined as follows:

$$\begin{aligned} I(g) &= g.PS() - g.boundary().PS() \\ F(g) &= g.boundary().PS() \end{aligned}$$

$$\forall g_i, g_j \in \text{self.element} \quad (g_i \neq g_j \Rightarrow ((I(g_i) \cap I(g_j) = \emptyset) \wedge (F(g_i) \cap F(g_j) \neq \emptyset) \Rightarrow |F(g_i) \cap F(g_j)| < \infty))$$

Notice that adjacency on a section of the boundary is not supported since the two surfaces would be representable with a single surface of type *GU_CPSurface2D*.

Specialization of inherited properties

- **boundary()**
it returns an aggregate of type *GU_CXRing2D*, the components of which are rings that represent the external and internal boundaries of all aggregate component surfaces.
`self.boundary()=self.element.boundary()`
- **dimension**
`self.dimension() = 2`
- **isCycle**
A planar surface is not closed by definition.
`self.isCycle() = false`
- **isSimple**
The individual surfaces of components are simple by definition and the definition of constraints set by the type guarantee the property of simplicity of the aggregate.
`self.isSimple = true`

Figure 4.7 shows complex surfaces made up of two disjoint elementary surfaces (case (a)), adjoining at one point (case (b)) and at two points (case (c)). Figure 4.8 shows two surfaces that are not representable as complex surfaces, but as elementary surfaces (case (b)) and as a generic aggregate (case (a)) in which the linear section is a curve that is distinct from the two surfaces.

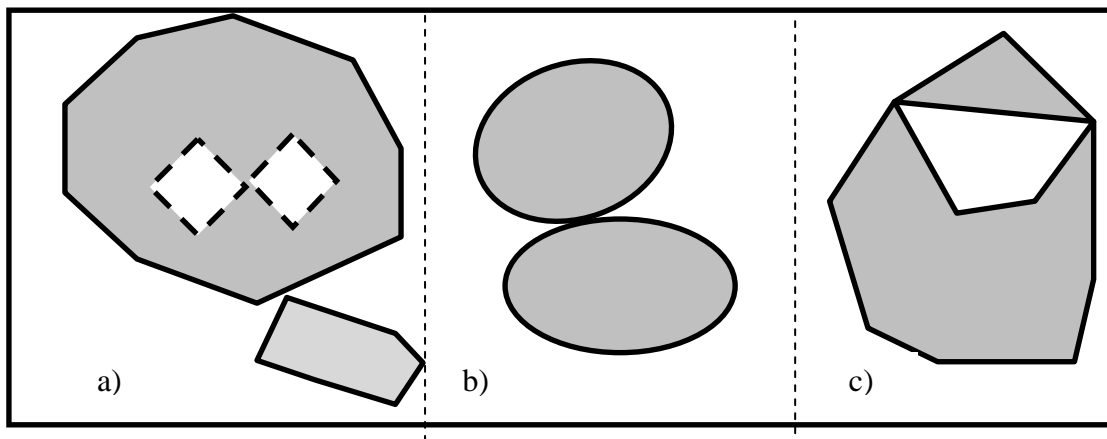


Figure 4.7 – Examples of complex surfaces (*GU_CXSurface2D*) composed of two elementary surfaces.

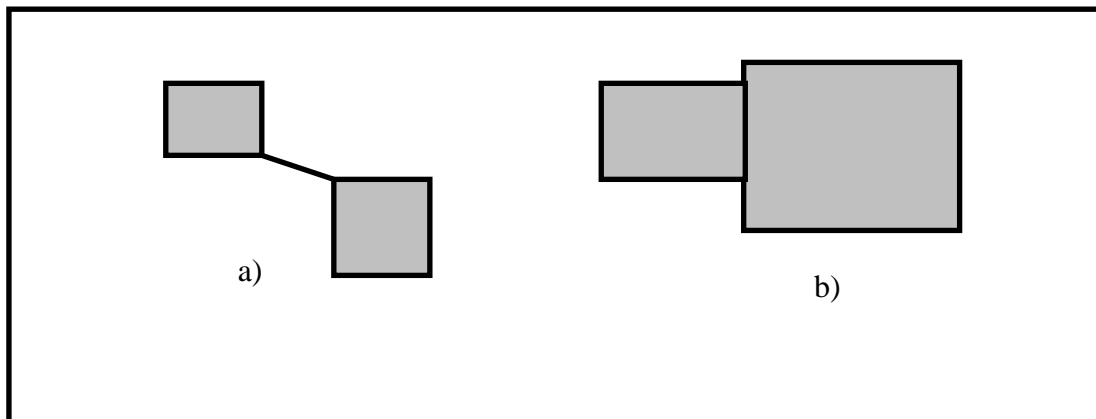


Figure 4.8 – Examples of geometries not describable as *GU_CXSurface2D*

4.2.15 GU_CPSurfaceB3D/GU_CXSurfaceB3D (Composite/Complex Surface Boundary 3D)

GeoUML models surfaces in 3D space through the concept of a surface with boundary in 3D (types *GU_CPSurfaceB3D* e *GU_CXSurfaceB3D*). These two types describes the surface through two geometric attributes interconnected by a constraint:

- attribute “B3D” which for both types describes the real boundary of the surface in 3D space; this boundary may be composed of multiple rings and is defined using an aggregate of rings of type *GU_CXRing3D*;
- the “surface” attribute which describes the planar projection of the surface in 2D space using a *GU_CPSurface2D* primitive surface in type *GU_CPSurfaceB3D* and a *GU_CXSurface2D* surface aggregate in type *GU_CXSurfaceB3D*.

The constraint joining the two attributes implies that the boundary of the projected surface coincides with the planar projection of the 3D boundary.

Specific attributes of type *GU_CPSurfaceB3D*

- **surface:** *GU_CPSurface2D*
- **B3D:** *GU_CXRing3D*;

Specific attributes of type *GU_CXSurfaceB3D*

- **surface:** *GU_CXSurface2D*;
- **B3D:** *GU_CXRing3D*;

Constraint on attributes

self.B3D.planar().PS() = self.surface.boundary().PS()

Specialization of inherited properties

The properties defined on *GU_Object* are significant for component geometries of a surface of this type, but not for the composed geometry, therefore the functions *boundary()*, *coordinateDimension()*, *dimension()*, *isCycle()*, *isSimple()*, and *planar()* will assume the null value.

Comment and Example

The constraint on attribute restricts the configuration of rings which describe the B3D curve to only those which, when projected, remain rings that satisfy the constraints set by the surface attribute. B3D surfaces have many possible applications, being able to display areal objects considering them in three-dimensional space as simple rings (i.e. without a precise determination of the three-dimensional surface delimited by the ring itself), while at the same time defining many additional properties with reference to surfaces, such as the coverage of an area, adjacency, containment of other geometric objects, etc., referring to 2D surfaces delimited by projections of such rings.

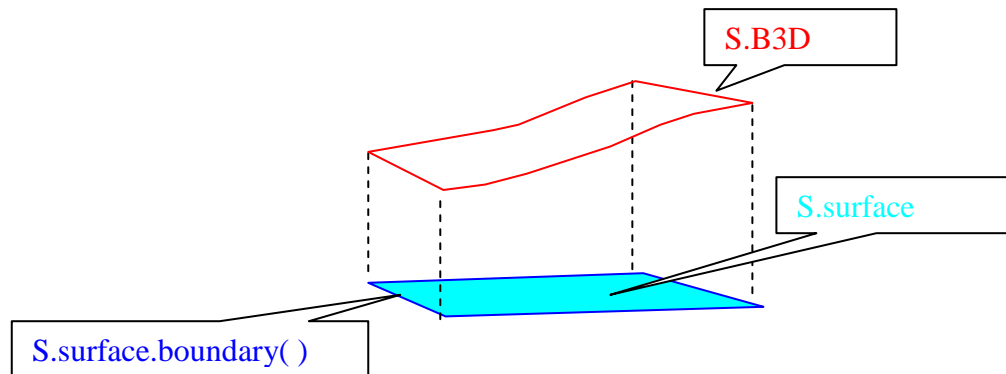
An example declaration in GeoUML with reference to this geometric type is the following one:

```
class Lake (LAK – 0802)
  attributes
    class spatial components
      080201 - extension: GU_CPSurfaceB3D;
  ...
```

Note that the type is a composition of other types, therefore it has no associated properties as a whole. In spatial relations and in all expressions of constraints, reference must be made to component attributes. This implies that when referring to an attribute of type

*GU_C*SurfaceB3D* “.surface“ or “.B3D“ must be added, depending on which of the two components is considered; with reference to the previous example, you should write as follows:
“Lake.extension.surface” or “Lake.extension.B3D”
in order to express relations or constraints that refer to the spatial component of the Lake class.

The following figure shows an object of type *GU_CPSurfaceB3D*, with its two components highlighted; it also shows the use of the function `boundary()` with reference to the “surface” component of the object.



Final observation: it is stressed that the type “surfaceB3D” is defined at conceptual level; therefore there may be Implementation Models which do not require an explicit representation of the two components “surface” and “B3D”.

4.2.16 *gUnion* (geometric union) and *gIntersection* (geometric intersection) functions

The operation *gUnion* (to distinguish it from a union between objects) applies to two objects belonging to the defined geometric types (or to one of the two components of a *surfaceB3D*) and produces the set of points obtained from the set theory union of point sets from the objects involved. This point set is then associated to an object or to an aggregate of objects of one of the types defined in the GeoUML geometric model. The same applies to the operation *gIntersection*, that is, it applies to two geometric objects and produces the set of points obtained from the set theory intersection of point sets from the involved objects. Like with *gUnion*, the result is the associated to an object of one of the GeoUML geometric types.

Table 4.1 shows the types that can be involved in the operations and Tables 4.2(a) and 4.2(b) show the subtypes of the *GU_Object* type produced by the *gUnion* and *gIntersection* operations, respectively. The tables only show the type codes for operands and for the results. Note that the objects involved in the operation and the result must belong to the same space (2D or 3D).

Certain cells in the tables indicate the possibility of generating different types of results, in particular the generic aggregate type when the *gUnion* (or *gIntersection*) is made between objects of a different dimension.

GeoUML Model
Geometric Model and OCL Constraints Templates

Type code	Typo
∅	Empty set
P	<i>GU_Point*D</i>
C	<i>GU_CPCurve*D, GU_CPSimpleCurve*D, GU_CPRing*D,</i>
S	<i>GU_CPSurface2D</i>
MP	<i>GU_CXPoint*D,</i>
MC	<i>GU_CXCurve*D, GU_CXRing*D, GU_CNCCurve*D</i>
MS	<i>GU_CXSurface2D</i>
A	<i>GU_Aggregate*D</i>

Table 4.1. Types of operands in gUnion and gIntersection

<i>gUnion(a,b)</i>								
b	∅	P	C	S	MP	MC	MS	A
a	∅	P	C	S	MP	MC	MS	A
∅	∅	P	C	S	MP	MC	MS	A
P	////////	P, MP	C, A	S, A	MP	MC, A	MS, A	A
C	////////	////////	C, MC	S, A	C, A	C, MC	MS, A	A
S	////////	////////	////////	S, MS	S,A	S, A	S, MS	A
MP	////////	////////	////////	////////	MP	MC, A	MS, A	A
MC	////////	////////	////////	////////	////////	C, MC	MS, A	A
MS	////////	////////	////////	////////	////////	////////	S, MS	A
A	////////	////////	////////	////////	////////	////////	////////	A

(a)

<i>gIntersection(a,b)</i>								
b	∅	P	C	S	MP	MC	MS	A
a	∅	P	C	S	MP	MC	MS	A
∅	∅	∅	∅	∅	∅	∅	∅	∅
P	//////	P	P	P	P	P	P	P
C	//////// ////////	////// //////	P, MP, C, MC, A	P, MP, C, MC, A	P, MP	P, MP, C, MC, A	P, MP, C, MC, A	P, MP, C, MC, A
S	////////	//////	//////////	any	P, MP	P, MP, C, MC, A	any	any
MP	//////// ////////	////// //////	//////////	//////////	P, MP	P, MP	P, MP	P, MP
MC	//////// ////////	////// //////	//////////	//////////	////////	P, MP, C, MC, A	P, MP, C, MC, A	P, MP, C, MC, A
MS	////////	//////	//////////	//////////	////////	//////////	any	any
A	////////	//////	//////////	//////////	////////	//////////	//////////	any

(b)

Table 4.2 Types of object produced by gUnion (a) and gIntersection (b).

4.3 Topological relations

In order to describe the spatial relations between objects, particularly when specifying geometric integrity constraints in a GeoUML schema, it is necessary to use a set of topological relations for reference.

GeoUML topological relations are defined using the concepts of internal part, boundary and external part of a geometric object; given a geometric object *a* of type *GU_Object* the following are defined:

1. internal part of *a*, denoted as $I(a)$: it is the point set

$$a.PS() - a.boundary.PS()$$
 (it is the set of points of an object that do not belong to its boundary)
2. external part of *a*, denoted as $E(a)$: it is the set of points from the space that do not belong to the object itself.

The fundamental set of topological relations used in GeoUML, denoted as REL_{topo} , is made up of the relations: Disjoint (DJ), Touches (TC), In (IN), Contains (CT), Equals (EQ) e Overlaps (OV). This set possesses the following characteristics:

- its constituent relations are mutually exclusive, that is, where the relation *R* is valid between two geometric objects, no other relation of that set is valid
- the set is complete, that is, given two geometric objects in a certain spatial situations, the set will always have a relationship that is true in that situation
- relations apply to objects of the same type or of different types.

Relations of the REL_{topo} set do not specify the dimension of the result and are applicable to all GeoUML geometric objects except the generic aggregate (as it does not have the boundary concept defined); in the case of B3D surfaces, these must be applied by specifying one of the attributes that make up the type. Where topological relations are applied to aggregate types, there do not involve the components of the aggregate individually, but apply to the aggregate point set interpreted as the result of the union of points from the components (e.g. the relation “Overlaps” is satisfied by an aggregate if at least one component of the aggregate satisfies it).

Mainwhile, REL_{topo} set relations are only comparable between objects defined in the same space (2D or 3D); comparison between objects defined in different spaces is not supported.

Definition of the set of relations REL_{topo}

With *a* and *b* as two geometric objects of any type with the exception of types *GU_Aggregate2D*, *GU_Aggregate3D*, *GU_CPSurfaceB3D* and *GU_CXSurfaceB3D*:

DJ: $a.Disjoint(b) \equiv_{def} (a.PS() \cap b.PS() = \emptyset)$

TC: $a.Touches(b) \equiv_{def} (I(a) \cap I(b) = \emptyset) \wedge (a.PS() \cap b.PS() \neq \emptyset)$

IN: $a.In(b) \equiv_{def} (a.PS() \cap b.PS() = a.PS())$
 $\wedge (a.PS() \cap b.PS() \neq b.PS()) \wedge (I(a) \cap I(b) \neq \emptyset)$

CT: $a.Contains(b) \equiv_{def} b.in(a)$

EQ: $a.Equals(b) \equiv_{def} (a.PS() \cap b.PS() = a.PS())$
 $\wedge (a.PS() \cap b.PS() = b.PS())$

OV: $a.Overlaps(b) \equiv_{def} (I(a) \cap I(b) \neq \emptyset)$
 $\wedge (a.PS() \cap b.PS() \neq a.PS())$
 $\wedge (a.PS() \cap b.PS() \neq b.PS())$

The minimum complete set REL_{topo} is expanded by the relations `Intersects` and `Cross` (between curves); they can be defined from the others, but are commonly used, so they are added for convenience:

INT: `a.Intersects(b) \equiv_{def} \neg a.Disjoint(b)`

CR: `a.Cross(b) \equiv_{def} a.Overlaps(b) \wedge (a.PS() \cap b.PS()).dimension()=0`

Comment

Notice that:

- The relation DJ impedes common points between objects, while all others require that the two objects have common points.
- Where common points are not internal points of objects, then the relation is TC. This definition considers not only cases of clear adjacency, with only points of the boundary shared, but also more complex cases in which the boundary points of one object are also internal points of another. This implies, for example, that a curve contained in the boundary of a surface is a possible case for the relation TC. The relation TC will always be false when both objects belong to `GU_Point*D`.
- The containment IN (CT) corresponds intuitively to the concept of set theory containment, except where an object is contained in the boundary of another, as described in the previous point (TC relation) or equal to another (EQ relation).
- In case of OV relation, the two objects have internal common points (therefore they do not satisfy the TC relation), but are not in IN(CT) or EQ relation (thus both objects have a part which is outside the part they share in common). The relation OV is therefore false when one or both objects are points.
- The relation CR is a specialization of the relation OV, which applies only to `GU_C*Curve*D` type objects and verifies that the dimension of intersection is zero (i.e. it is a finite set of points).
- the relations OV, DJ, CR, EQ, TC are symmetrical (e.g., `a.Touches(b)` is the same as `b.Touches(a)`);
- the relation DJ between two objects is always true where the geometry of at least one of the two objects is empty, while the other topological relations are always false in the presence of at least one empty geometry.

The relations of the set REL_{topo} (except Equals and Contains) are illustrated in Figure 4.9, in which each column shows the same topological relation applied to different types of objects and each row shows the different relations that apply to the same pair of object types.

GeoUML Model

Geometric Model and OCL Constraints Templates

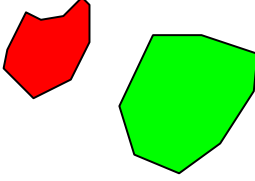
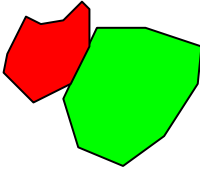
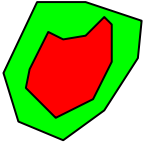
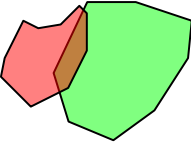
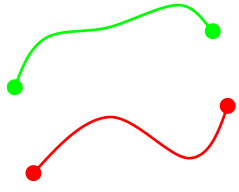
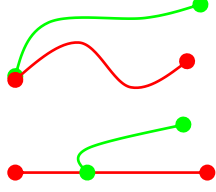
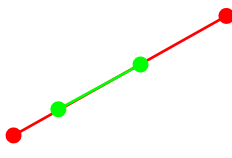
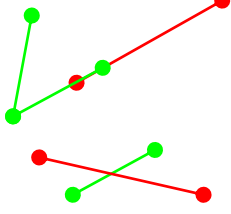
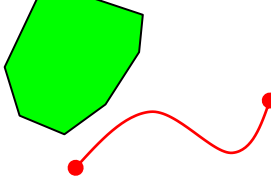
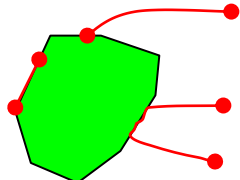
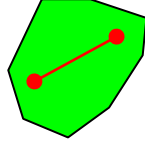
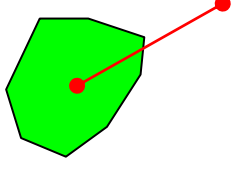
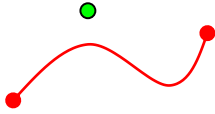

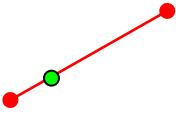
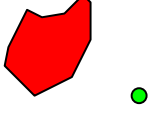



			
DISJOINT	TOUCHES	IN	OVERLAPS
			
DISJOINT	TOUCHES	IN	OVERLAPS
			
DISJOINT	TOUCHES	IN	OVERLAPS
			
DISJOINT	TOUCHES	IN	
			
DISJOINT	TOUCHES	IN	
			
DISJOINT			

Figure 4.9 – Example of topological relations on different types of geometric objects.

5 Geometry-dependent attributes

5.1 Introduction

Geometry-dependent attributes are attributes whose value is a function of the points belonging to a geometric attribute of a class object. The geometry-dependent attribute has three variants: segmented attributes, subregions attributes, and events. Defined below are the segmented attribute and subregions attribute dependent on linear and areal geometry, respectively, and the events attribute dependent on both geometries.

Comment and Example

For example, if we consider the “location” attribute of a street with a “route” linear attribute, the value of this attribute not only depends on the street in question but also on the point of the route taken into consideration. The points of the route can be grouped into zones with a uniform value of the “location”, called *segments*.

If we consider the same example but with an areal representation of the street, the points with a uniform “location” value constitute *subregions*.

The following definitions apply to different geometric types; therefore the formation is factorised using the following conventions:

- The asterisk character in a geometric type name indicates all possible values that could be in that position (for example GU_Object*D means any object in two or three dimension)
- The indication of a geometric type in a definition means that the same definition applies to that type and to any of its specialisations.

5.2 Segmented attribute

Given a class X containing a linear geometric attribute g, it is possible to define for g a segmented attribute A of domain D_A which describes a property dependent on the geometry g. This definition is based on the keyword *segments On* inserted in the section *attributes of this spatial component* of the reference spatial component, as shown in the following example.

```
class X (abbreviations)
    ....
    class spatial components
        g: GU_C*Curve*D;
        attributes of this spatial components
            A [min..max]: DA segments On g;
    ...
```

The segmented attribute has a unique name among the attributes of the spatial component on which it is defined, in addition to the code and optional alphanumeric code. The domain D_A of the segmented attribute may be a base domain, an enumerated domain, or a hierarchical enumerated domain without additional base domain attributes (the DataType domain is excluded).

The definition of the meaning of these types of attributes is based on replacing the declaration:

```
A [min..max]: DA segments On g;
```

with the following two functions:

```
ValuesOf_A(p: GU_Point*D): Set(DA)  
SegmentsOf_A(cond: String): Set(GU_CXCurve*D)
```

These two functions can be assumed implicitly declared and can be used to specify certain additional properties of classes, as occurs when specifying integrity constraints; their replacement makes it possible to proceed as though the schema in the previous example were written as follows:

Representation in basic GeoUML + methods

```
class X (alphanumericCodeX - codX)  
  attributes  
  class spatial components  
    g: GU_C*Curve*D;  
  methods  
  ValuesOf_A(p: GU_Point*D): Set(DA)  
  SegmentsOf_A(cond: String): Set(GU_CXCurve*D)
```

The function `ValuesOf_A(p)` returns the value of the segmented attribute *A* on a specific point *p* of the geometric attribute *g*. It receives as parameter a point *p* defined in the same space (2D/3D) of the attribute *g* and, where the point belongs to *g*, the function returns a value of the domain D_A where the cardinality is 1..1, or a set of values where the maximum cardinality is * (it is not possible to assign values other than 1 and * to maximum cardinality). Finally, in case of an absent value (optional attribute, i.e. minimum cardinality = 0) it returns the *null* value; the *null* value is also returned where the point does not belong to *g* or when *g* is *null*.

A **segment** is a geometric object of the type `GU_CXCurve*D` (a segment cannot contain isolated points) of the same dimension as the geometric attribute *g* and is associated with a value $v \in D_A$; therefore a segment is made up of the union of all points *p* of *g* for which `ValuesOf_A(p) = v`.

From this definition it is deduced that:

- a segment generally corresponds to a complex and non-connected curve (the segment may contain bifurcations due to self-intersections or intersections between components of the geometry);
- in the case of optional attributes, a segment may exist that contains all points in which the attribute has the *null* value;
- two different segments may overlap, for example, when a portion of the geometry *g* shares two or more attribute values (only where the maximum cardinality is *) or they may intersect (in the presence of self-intersections or intersections between components of the geometry);
- the union (`gUnion`) of all point sets from the segments defined on *g* corresponds to the geometry of *g*.

The function `SegmentsOf_A` returns the *null* value when the value of the geometric attribute *g* is *null*. The function `SegmentsOf_A(selection condition)`, given a particular selection condition, returns the set of segments that satisfy that condition; the selection clause is a propositional formula of the type: `[not](α_1 opLogico ... α_1 ... opLogico α_n)` with `opLogico` \in {AND, OR} and where α_i is an atomic formula of the type `(A op cost)`, `(A = null)` or `(A = not null)` where *A* is the segmented attribute, `op` \in {=, <>, >, \geq , <, \leq } and `cost` is a value other than *null*. An empty condition corresponds to the

GeoUML Model

Geometric Model and OCL Constraints Templates

constant true, and thus all segments defined on the geometry g will be returned, while a condition of always false will return the empty set.

It is also possible to defined the segmented attribute on the boundary of a surface of types $GU_C*Surface2D$ e $GU_C*SurfaceB3D$; in the first case, a segmented attribute would be defined on the 2D boundary of the geometric attribute, while for the surface with 3D boundary there are two possibilities:

- the attribute is defined on the curve (type B3D component) which represents the boundary of the surface in 3D space and is described as a segmented attribute on the 3D boundary,
- the attribute is defined on the boundary of the surface (type surface component) obtained by the projection of the type's B3D component in 2D space, and is described as a segmented attribute on the 2D boundary of the geometry.

In these cases, the attribute is syntactically expressed as shown below, while its meaning is unchanged; note that the functions `ValuesOf_A()` and `SegmentsOf_A()` are defined in the template on the specific 2D or 3D boundary of the geometric attribute in question.

Variant on surface boundary

```
class X
...
class spatial components
g: GU_C*Surface2D or GU_C*SurfaceB3D;
attributes of this spatial component
A[min..max]: DA segmented On 2D|3D boundary of g;
```

Comment and Example

With reference to the above example, the following definition associates with streets a segmented attribute on its route representing the street *vertical position*.

class Street (STR – 0501)

attributes

class spatial components

050101 - route: GU_CPCurve3D;

attributes of this spatial component

050102 - verticalPosition: Enum VP_TYPE segmented On route;

domain VP_TYPE (TVP – 9977)

01 on ground surface

...

N.B.: in an Implementation Model, a segmented attribute may be created both by fabricating geometries that from a single segment and associating them with the attribute value, and by using a curved abscissa to indicate the portions of the spatial component on which the attribute has a certain value. The physical structure of the Data Product will contain adequate information on the type of implementation selected.

The functions used here to define the semantics of the segments, which will also be used in the following chapters (such as to express constraints), can generally be implemented on both of these implementations; therefore a Data Product has no need to contain the implementation of the functions.

These consideration are also valid for events and sub-region attributes; however, for the latter, the only form of implementation is through the fabrication of geometries, not through a curved abscissa.

5.3 Events attribute

The definition of an events attribute follows similar rules to that of segmented attributes.

Given a class X containing a linear geometric attribute g or areal geometric attribute (excluding the surfaces GU_C*SurfaceB3D), it is possible to define for g an events attribute A of the domain D_A which describes a property dependent on the geometry g. This definition is based on the keyword *Events On* inserted in the section *attributes of this spatial component* of the reference spatial component, as shown in the following example.

```

class X
...
  class spatial components
    g: GU_C*Curve*D or GU_C*Surface2D;
    attributes of this spatial component
    A[0..max]: DA Events On g;
  ...

```

The events attribute has a unique name among the attributes of the spatial component on which it is defined, in addition to the code and optional alphanumeric code. The domain D_A may be a base domain, and enumerated domain, or a hierarchical enumerated domain without additional base domain attributes (the DataType domain is excluded).

An event is a point of the geometry to which a value $v \in D_A$ is associated; not all points of the geometry have an associated event (minimum cardinality is always 0) and there may be points that are associated with multiple events (maximum cardinality *).

As with segments, the definition of the meaning of these types of attributes is based on replacing the previous declaration with a function, like in the following example:

Representation in basic GeoUML + methods

```

class X (alphanumericCodeX - codX)
  attributes
  class spatial components
    codg - g: GU_C*Curve*D or GU_C*Surface2D;
  methods
  EventsOf_A(cond: String): Set(GU_CXPoint*D)

```

The function EventsOf_A(cond), given a particular selection condition, returns the set of points that satisfy that condition; the selection clause is a propositional formula of the type [not](α_1 opLogico ... α_i ... opLogico α_n) with opLogico ∈ {AND, OR} and α_i is

an atomic formula of the type $(A \text{ op } \text{cost})$, where A is the events attributes, $\text{op} \in \{=, <>, >, \geq, <, \leq\}$ and cost is a value other than *null*.

An empty condition corresponds to the constant true, thus in this case the function will return all points defined on the geometry g on which an event is defined.

The function `EventsOf_A()` returns the null value when the value of the geometric attribute g is *null*.

5.4 Subregions attribute

Like with the segmented attribute, the subregions attribute associates a value of a domain to different subregions of an areal geometric attribute.

Comment and Example

An example in this case would be a “street” class with an areal geometric attribute “extension” and a “vertical position” attribute.

In this example, we can define a subregions attribute “vertical position” which associates the value of the “vertical position” to various points of the extension of the street; a “subregion” is a portion of the extension to which a particular attribute value is homogeneously associated.

Given a class X containing an areal geometric attribute g , it is possible to define for g a subregions attribute A of the domain D_A that describes a property dependent on the geometry of g . This definition is based on the keyword *Subregions On* inserted in the section *attributes of this spatial component* of the reference spatial component g , as shown in the following example.

```
class X (alphanumericCodeX - codX)
  ...
  class spatial components
    g: GU_C*Surface2D or GU_C*SurfaceB3D;
    attributes of this spatial component
      A[min..max]: DA Subregions On g;
    ...
```

The subregions attribute has a unique name among the attributes of the spatial component on which it is defined, in addition to the code and optional alphanumeric code. The domain D_A may be a basic domain, an enumerated domain, or a hierarchical enumerated domain without additional base domain attributes (the `DataType` domain is excluded)

The meaning of the subregions attribute is defined using the same substitution rules using two methods as done in the case of segmented attributes, as in the following example:

Representation in basic GeoUML + methods

```
class X (alphanumericCodeX - codX)
  attributes
  class spatial components
    codg - g: GU_C*Surface2D or GU_C*SurfaceB3D;
  methods
    ValuesOf_A(p: GU_Point2D): Set(DA)
    SubregionsOf_A(cond: String): Set(GU_CXSurface2D)
```

The function `ValuesOf_A(p)` returns the value of the subregions attribute in a given point p of the geometric attribute g . It receives as parameter a point p in 2D space and, where the point

belongs to the geometric attribute g , the function returns a value of the domain D_A where the cardinality is 1..1, or a set of values where the maximum cardinality is * (it is not possible to assign values other than 1 and * to maximum cardinality). Finally in the case of an absent value (optional attribute, i.e. minimum cardinality = 0) it returns the *null* value; the *null* value is also returned when the point does not belong to the geometry or when the geometry is *null*.

A **subregion** is a geometric object of the class `GU_CXSurface2D` (a subregion cannot contain isolated points or curves) and is associated with a value $v \in D_A$, and therefore a subarea is made up of the union of all points p of g for which `ValuesOf_A(p) = v`.

From this definition it is deduced that:

- a subregion generally corresponds to a complete, unconnected surface;
- in the case of optional attributes, a subregion may exist that contains all points in which the attribute has the *null* value;
- two different subregions may overlap, for example, when a portion of the geometry g shares two or more attribute values (only where the maximum cardinality is *);
- the union (`gUnion`) of all subregions defined on g corresponds to the geometry of g .

The function `SubregionsOf_A()` returns the null value when g is *null*.

The function `SubregionsOf_A(selection condition)` given a particular selection condition, returns the set of subregions that satisfy that condition; the selection clause is a propositional formula of the type `[not](α_1 opLogico ... α_i ... opLogico α_n)` with `opLogico` \in {AND, OR} and α_i is an atomic formula of the type `(A op cost)`, ($A = \text{null}$) or `(A = not null)` where A is the subregion attribute, `op` \in {=, <>, >, \geq , <, \leq } and `cost` is a value other than *null*. An empty condition corresponds to the constant true, and thus all subregions defined on g will be returned, while a condition of always false will return the empty set.

Comment and Example

The following example in GeoUML is exactly the same as that shown above, but is developed by interpreting the vertical position as subregions of an areal representation of the street.

class Street (STR – 0501)

attributes

class spatial components

050101 - extension: GU_CPSurface2D;

attributes of this spatial component

05010101 - VerticalPosition: Enum T_VP Subregions On extension

domain T_VP (TSED – 9977) ...

Subregions and B3D surfaces

When the spatial component of an object is of the type `GU_*SurfaceB3D`, the subregions are defined on the “surface” component, thus making them 2D surfaces as shown in Figure 5.1.

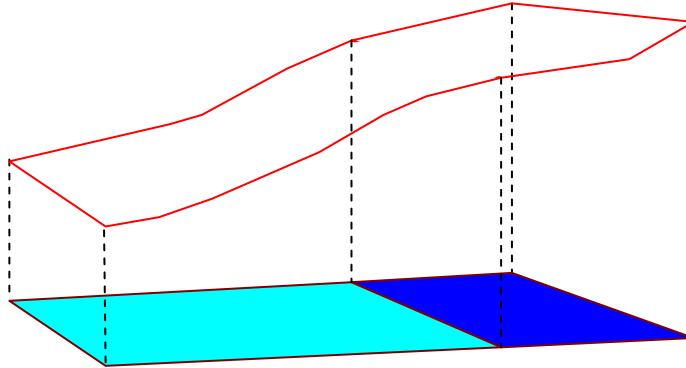


Figure 5.1 – A B3D surface with 2 subregions.

Notice that the representation shown is at the conceptual level and serves to uniquely determine the interpretation of objects, but also allows to create different implementation models.

6 Spatial integrity constraints

6.1 Introduction

Spatial integrity constraints express conditions that must be satisfied by the spatial components of the class instances to which they refer.

Constraints are an important aspect of a Data Product's conformity with a specification; therefore their meaning must not be ambiguous. For this reason, a formal definition is given of the meaning of constraints based on a set of rules for translation into OCL (Object Constraint Language of UML): **in situation where the intuitive meaning of a constraint appears ambiguous, reference must be made to its definition to determine its correct interpretation.**

Since the rules for translation into OCL are numerous and difficult to read, in this chapter, constraints are illustrated in natural language to facilitate an interpretation of their general meaning and expressivity and a clear understanding of the subsequent chapters.

The rules for translation into OCL are set out in this chapter for illustration purposes only for the first type of constraint, while all others are described in Annex A.

There are two families of spatial integrity constraints: topological constraints and part-whole constraints.

6.2 Topological constraints

Topological constraints use topological relations from the set REL_{topo} , defined on all instanceable geometric objects of GeoUML, providing they have a defined boundary method; therefore generic aggregate attributes cannot be involved in the constraints as topological relations do not apply to them.

There are three categories of topological constraints:

1. existential constraints
2. union constraints
3. universal constraints

For each of these categories there are a base version and different variants for describing more detailed conditions, supplementing the base version with a combination of the following elements:

- selection of objects of the classes involved in the constraint
- application of the functions `boundary()` and `planar()` to the spatial components of the objects involved in the constraint
- use of an association between constrained and constraining classes in the formulation of the constraint
- use of segmented, events, and subregions attributes in the constraint

The presentation of topological constraints is as follows:

- first of all, the existential topological constraint in the base version is illustrated in detail,
- this is followed by several general rules that apply to the expression of constraints,
- a definition is then given, with reference to the existential constraint in its base version, of the formulation method through translation into OCL which is applied extensively in Annex A,
- the variants of the existential constraint are then defined,
- this is followed by a definition of the constraint on union, with variants the same as the existential constraint,
- finally, a definition is given of the universal constraint, with variants the same as the existential constraint.

6.2.1 Basic existential topological constraint

The basic existential topological constraint requires, for each object x of constrained class X the existence of at least one object y of constraining class Y , such that a geometric attribute f of y is found in a particular topological relation with a geometric attribute g of x .

Example

Description of the constraint in natural language:

for each *RoadElement* (constrained class) there must be a *RoadArea* (constraining class) whose *extension* (spatial component of the constraining class) contains the *route* (spatial component of the constrained class) of the *RoadElement*.

Textual definition of the constraint:

constraint RoadElement.route (IN) exists RoadArea.extension

this constraint operates on classes and spatial components which must be defined as follows:

```
class RoadArea (...)  
  attributes  
    class spatial components  
    ... extension: GU_CPSurface2D  
class RoadElement (...)  
  attributes  
    class spatial components  
    ... route: GU_CPCurve2D
```

The constraint is thus characterised by the following aspects:

- The **constrained class** and its geometric attribute (*RoadElement* and *extension* in the example)
- The **constraining class** and its geometric attribute (*RoadArea* and *route* in the example)
- The **topological relation** (IN in the example),

Graphic representation of the constraint:

The graphic representation of the same example is shown in Diagram 6.1, which substantially contains the same elements as the textual definition, but for which a name must be given to the individual constraint (in the figure this is *VT_RoadElement*).

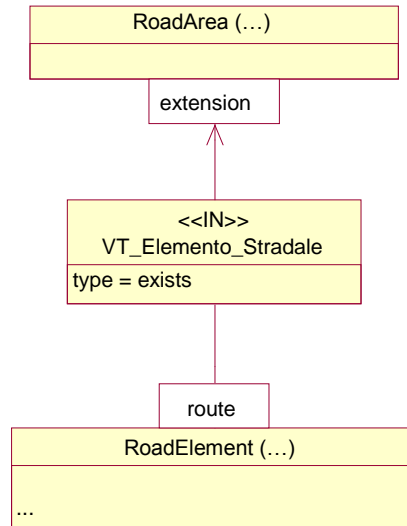


Diagram 6.1 – Example of existential topological constraint

6.2.2 General rules for constraint formulation

For all constraints that can be formulated in GeoUML, the following general rules apply:

- **Uniqueness of space for the application of relations:** the types of all geometries involved in a constraint must belong to the same space (2D or 3D) as topological relations are not defined among objects belonging to different spaces (in some cases, to satisfy this rule we will see the *planar()* function used, which changes the embedding space of a particular geometry;
- **Disjunction of topological relations:** the topological relation of a constraint can generally be replaced by a disjunction of elementary topological relations, that is, by different relations placed in OR between them (e.g. “DJ or TC”, denoted by “DJ|TC”).
- **Inheritance hierarchy:** in the presence of inheritance hierarchies, the definition of a constraint between two classes implies its implicit application to all subclasses (direct and indirect) of the constrained class and that the constraint satisfaction on each constrained object involves the objects of the constraining class and those of all its subclasses (direct and indirect). Additionally, the definition of the constraint may refer to constrained or constraining class properties or to those inherited from the ancestors of the constrained or constraining classes;
- **Self-ring:** when a constraint involves the same class, both as constrained class and as constraining class (self-ring), the set of constraining objects that are taken in account in order to satisfy the constraint of a certain object O is made up of all objects of the class excluding *the object O itself*;
- **Surfaces with 3D boundary:** when one or both geometric attributes f and g are of the type $GU_C^*SurfaceB3D$ it is necessary to specify the component considered in the constraint, that is, the $B3D$ attribute or the *surface* attribute;
- **Abbreviations:** in order to simplify the formulation of constraints, the alphanumeric class code may be used instead of its full name, while the class prefix may be omitted from in front of the attribute names where the class is that of the current context;

- **Limit on the use of attributes:** constraints may not refer to attributes of an association or to additional base attributes present in the hierarchical enumerated domain of an attribute.

Additionally, the rules for the graphic representation of a basic existential constraint apply to all constraints and all variants:

1. the constraint is represented by an arrow pointing from the constrained class to the constraining class
2. on both the constrained and the constraining class, the points where the arrow leaves or arrives show the spatial components to which the constraint refers (also applicable to this point are the selections and the functions used by the variants described below)
3. the arrow is documented by a rectangle which contains the topological relation (IN in the example), the name of the constraint (VT_RoadElement in the example), and the general constraint type (type=exists in the example).

6.2.3 Formal definition of the existential constraint using OCL translation rules

The formal definition of all constraints in Annex A is made up of the following parts:

1. *Constraint name*
2. *Definition of symbols*
3. *Constraint syntax in GeoUML*, including certain fixed keywords typical of that type of constraint (*constraint* and *exists* in the basic existential constraint) and certain variable parts, defining the classes and spatial components to which the constraint refers and which topological relations it applies.
4. *Corresponding OCL template*: a template is a function that has a name (in this case `ExistentialTopoConstraint`) and certain parameters (in this case `X`, `g`, `Y`, `f` and `DJ_R`). Such parameters correspond to the variable parts of the formulation of the constraint in GeoUML syntax and also appear in the OCL expression below. When the constraint is used in a GeoUML schema, these parameters are replaced with the name of the corresponding constructs of the schema on which the constraint is to act. Formulation of the constraint in OCL makes use of the `check` operation defined on the `GU_Object`, for verifying the satisfaction of a disjunction of topological relations r_1, \dots, r_n instead of a single topological relation; `check`, applied to an object `a` with reference to an object `b`, is formally defined as follows:

$$a.\text{check}(\{r_1, \dots, r_n\}, b) \equiv_{\text{def}} a.r_1(b) \vee \dots \vee a.r_n(b)$$

In the case of a basic existential constraint, the formalisation takes the following form:

Definition of symbols – Given two classes *X* and *Y* each containing at least one geometric attribute, respectively *g* and *f*, the topological existential constraint from *X* to *Y* based on the disjunction of relations $DJ_R = \{rel_1, \dots, rel_n\}$ is defined as follows:

Syntax:

```
constraint X.g (rel1 | ... | reln) exists Y.f
```

OCL template:

```
ExistentialTopoConstraint (X, g, Y, f, DJ_R) ≡  
context X  
inv: Y.allInstances ->  
    exists (a:Y | self.g.check(DJ_R, a.f))
```

GeoUML Model

Geometric Model and OCL Constraints Templates

Example application of the formal rule to the translation of a specific constraint

Applying this rule to the constraint from the previous example:

constraint RoadElement.route (IN) *exists* RoadArea.extension

we can see that we need to replace the list of parameters (X, g, Y, f, DJ_R) with the following (RoadElement, route, RoadArea, extension, IN) and we obtain the following constraint in OCL:

```
context RoadElement
inv: RoadArea.allInstances ->
    exists(a: RoadArea |
        self.route.check(IN, a.extension))
```

The constraint in this form could be inserted into an ISO standard Application Schema, provided that the check function is defined on the generic geometry.

6.2.4 Variants of the basic existential topological constraint

The variants of the existential topological constraint defined below can also be applied in combination.

6.2.4.1 Existential topological constraint with selections

This variant is used to select objects from the classes involved in the constraint.

Comment and Example

A selection on the constrained class limits the objects that must satisfy the constraint to those that satisfy the selection, rendering the constraint weaker; a selection on the constraining class reduces the number of objects that can be used to satisfy the constraint, rendering it stronger.

In the following example, the constraint from the previous example is restricted, requiring that only the *RoadElement* of a certain *type* must satisfy the constraint of being contained in a *RoadArea* object (the definition of the *RoadElement* class must also contain the *type* attribute):

constraint (RoadElement.tipo="T1")RoadElement.route
(IN) *exists* RoadArea.extension

Interpretation of the null value in the evaluation of constraints

GeoUML supports the optionality of the values of attributes/roles of a class and indicates the absence of value with the concept `null`.

The definition of constraints implies the possible selection of objects and/or segments (events, subregions) and the possible application of the boundary function, which can produce an empty set of objects and/or geometries.

The semantics of constraints in regard to this problem is governed by the following rules:

- any function (for example , `boundary()`) applied to a null value produces a null value as well;
- the interpretation of selection applied to classes (segments, etc.) in a constraint refer to the standard semantic SLQ92, including for null value interpretation; it should be noted that the presence of a null value when evaluating a condition may generate an “unknown” result (neither true nor false); in such cases, SQL forces a false evaluation and does not select the object. The same condition is verified in the selection of segments (events, subregions, etc.);
- geometries (of both the constrained and constraining classes) are used in the evaluation of a constraint only where they contain a non-null value; a constraint therefore establishes a condition that must be satisfied only by geometries that are actually available (even if empty) when evaluating the constraint.

6.2.4.2 *Existential topological constraints on the boundary or planar projection*

This variant is used for applying the functions BND (`boundary()`) and/or PLN (`planar()`) to spatial components involved in the constraint. These functions can also be applied in cascade.

Example

Consider the example from the previous paragraph with the following variation: assuming that the spatial component *route* of the class *RoadElement* has a geometry in 3D; in this case, the `planar()` function must be applied to the constraint to satisfy the general rule that topological relations are applied to geometries embedded in the same space:

constraint (RoadElement.type="T1")RoadElement.route.PLN
(IN) exists RoadArea.extension

The graphic representation of the example is shown in Diagram 6.2. Notice that in the application of general rules of graphic representation, the selection, and the function *PLN* are indicated in the box dedicated to the spatial component.

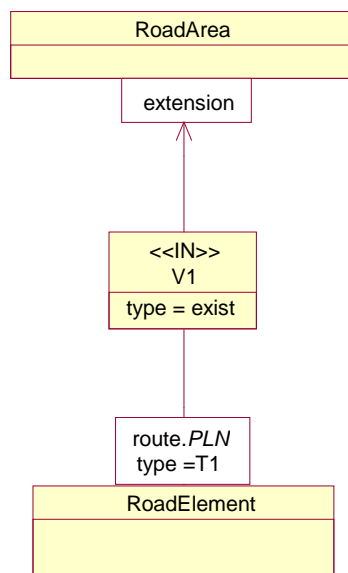


Diagram 6.2- Example of existential topological constraint on projection

6.2.4.3 *Topological constraint linked to an association*

With this variation, for purposes of satisfying the constraint, only the objects of the constraining class that are bound to the object of the constrained class by an association specified in the schema.

Example

The textual form of this variant is illustrated in the following example, which uses an association and the `planar()` function to show how different constraint variants can be combined.

constraint Road.route.PLN (IN) existx Road.municipalityOfRelevance.extension

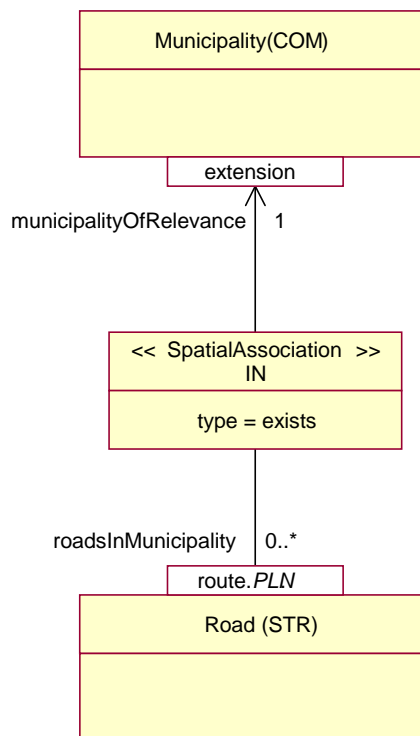
this constraint operates on classes and spatial components which must be defined as follows:

```

class Road (STR - 0501)
  attributes
    class spatial components
      route: GU_CPCurve3D
  roles
    municipalityOfRlevance [1..1] : Municipality

class Municipality (COM - 0101)
  attributes
    class spatial components
      extension: GU_CPSurface2D
  
```

The graphic form extends the general rules of representation, as it makes the representation of the constraint coincide with that of the association used, as follows:



6.2.4.4 Constraints on segmented or subregions attributes

A constraint can also refer to the geometry of attributes such as segmented, segmented on boundary, events or subregions attributes; in this case, the geometric attribute in the constraint specification must be replaced with the call of one of the functions to obtain the segments, events, or subregions according to the attribute type (for example, the function `SegmentsOf_A()` and `SubregionsOf_A()`, where `A` is the name of the segmented or subregions attribute); these functions may also include a selection of the segments or subregions that are involved in the constraint.

The following makes reference to the segmented attributes, but any constraint can be reformulated in a similar way for the segmented on boundary, events and subregions attributes.

Syntax:

Let X and Y be two classes and a and b two segmented attributes belonging to X and Y , respectively; the usage of the segmented attributes can be established on both classes, or on either the constrained class or the constraining class individually, as follows:

- a. constraint $X.SegmentsOf_a() (rel_1 | \dots | rel_n) \text{ exists } Y.SegmentsOf_b()$
this formulation requires that, for each segment of attribute a of class X , there exists a segment of the attribute b of class Y , satisfying the specified relations disjunction $rel_1 | \dots | rel_n$ between the two.
- b. constraint $X.SegmentsOf_a() (rel_1 | \dots | rel_n) \text{ exists } Y.f$
this formulation requires that, for each segment of attribute a of class X , there exists an instance of class Y with a spatial attribute f , satisfying the specified relations disjunction $rel_1 | \dots | rel_n$ between the two.
- c. constraint $X.g (rel_1 | \dots | rel_n) \text{ exists } Y.SegmentsOf_b()$
this formulation requires that, for the spatial component g of each instance of class X , there exists a segment of attribute b of class Y , satisfying the specified relations disjunction $rel_1 | \dots | rel_n$ between the two.

The graphic syntax of these variants is taken from the graphic syntax of the base constraint replacing the spatial component with the function `SegmentsOf_a()`, where a is the segmented attribute involved in the constraint, in the rectangles relative to the spatial component.

Example

The following constraint follows the structure from the above case (c), but using subregions instead of segments, applying a selection on the constrained class and a selection on the subregions attribute of the constraining class.

constraint (use = "road") Tunnel.Sup_verticalPosition.surface (CT) exists
RoadArea.SubregionsOf_verticalPosition(verticalPosition = "in tunnel")

Breakdown:

1. consider an instance T of the class *Tunnel* such that its use attribute = “road” (selection on constrained class)
2. consider its spatial component *Sup_verticalPosition*, which is surface with 3D boundary, taking the *surface* component of that value, which is on the 2D plane, and call that surface TS
3. there must exist a subregion SA of the subregions attribute *verticalPosition* of an instance of the class *RoadArea* with the value “in tunnel” such that GS contains SA

6.2.5 Topological Constraint on union

The topological constraint on union refers to the union (*gUnion*) of the spatial components of all instances of the constraining class *Y*, rather than requiring the existence of an individual instance that satisfies the constraint.

In other words, the topological relation is verified with respect to the geometry obtained from the union of values from the spatial components of all instances of *Y*.

For topological constraint on union, there are also variants presented for the existential topological constraint, that is: version with selection, version referring to boundary() and planar(), version referring to an association and version referring to segments, events and subregions attributes.

Comment and Example

An example of the constraint in text form is as follows:

constraint RoadElement.route (IN) *union* RoadArea.extension

this constraint operates on classes and spatial components which must be defined as follows

class RoadArea (...)

attributes

class spatial components

- extension: GU_CPSurface2D

class RoadElement (...)

attributes

class spatial components

- route: GU_CPCurve2D

This constraint requires, for each RoadElement, that the union of “extension” spatial component of all instances of the class RoadArea contains its route: the constraint is thus far weaker than the existential version discussed above.

The graphic representation of the constraint is the same as the existential constraint but replacing the wording “type=exists” with “type=union”.

6.2.6 Universal topological constraint

The universal topological constraint requires the presence of the topological relation between the constrained object and all instances of the constraining class. The universal topological constraint may also refer to the individual components of a surface with a 3D profile. There are also variants presented for the existential topological constraint, that is: version with selection, version with boundary() and planar() functions, version with segmented, events and subregions attributes, and the version referring to an association.

Comment and Example

The universal topological constraint is used almost exclusively with the spatial relation Disjoint, possible in disjunction with the spatial relation Touch. This occurs in the following example, which also illustrates the possibility – for all types of constraints – that the constrained class and constraining class coincide.

```
constraint RoadElement.route (DJ | TC)
                        forEach RoadElement.route
```

this constraint operates on classes and spatial components which must be defined as follows:

```
class RoadElement (ELESTR – 0504)
  attributes
  ...
  class spatial components
  .. .- route: GU_CPCurve3D
```

The graphic representation of the constraint is the same as the existential constraint but replacing the wording “type=exists” with “type=forEach”.

This example recalls the general rule on self-rings in constraints: when the constrained class coincides with the constraining class in the evaluation of an instance O of the constrained class, reference must be made to the instances of the same class, considered as constraining, but excluding the instance O, since the only satisfied spatial relation of a geometry with itself is *Equals*. In the previous example, this means that the route of an instance of RoadElement must be in a DJ|TC relation with the route of all other instances of the same class (but obviously not with itself).

6.2.7 Topological constraints with multiple constraining classes

All topological constraints can refer to multiple constraining classes, but it is important to ensure a careful interpretation of their meaning.

The syntax of the base form in this case is as follows:

Syntax:

constraint X.g... (rel₁ |...| rel_n) <type> (Y₁.f₁..., ..., Y_n.f_n...)

where <type> may be:

exists or forEach or union

The variants (selection, function boundary() and planar() and, reference to segmented, events and subregions attributes) may be applied separately to individual constraining classes.

In regards to the meaning, it is indispensable to consider the different behaviour of the quantifiers:

1. In the existential case (*exists*), an instance must exist in one of the constraining classes, for which the constraint is satisfied. Only one instance is necessary, thus the constraint might be not satisfied by any instances of one constraining class provided that there exists an instance in another constraining class that satisfies the constraint.
2. In the universal case (*forEach*) the constraint must be satisfied by all selected instances from all constraining classes.
3. In the case of constraint on union (*union*), the constraint must be satisfied by the geometric union of the specified geometric attribute of all instances from the constraining classes; in this case, the union of the geometries of the constraining classes is computed before the constraint is verified.

6.2.8 Disjunction of topological constraints

The disjunction of topological constraints allows us to indicate that, for each element of a constrained class, at least one of the constraints in the disjunction must be satisfied. In textual syntax, the constraints in disjunction are separated by the keyword OR and are preceded by the keyword *disjunction*.

Comment and Example

It must be ensured that *all constraint from the same disjunction refer to the same constrained class*. An example of disjunction of topological constraints in textual form is shown below. It requires that a *RoadElement* be disjoint from any other *RoadElement* or that its boundary belong to the set of *Junctions*:

disjunction RoadElement.route (DJ) *forEach* RoadElement.route
OR
RoadElement.route.*BND* (IN) *unione* Junction.position

From a graphic perspective, a disjunction of topological constraints is represented by conjoining the constraints in disjunction with a line, as shown in Diagram 6.6.

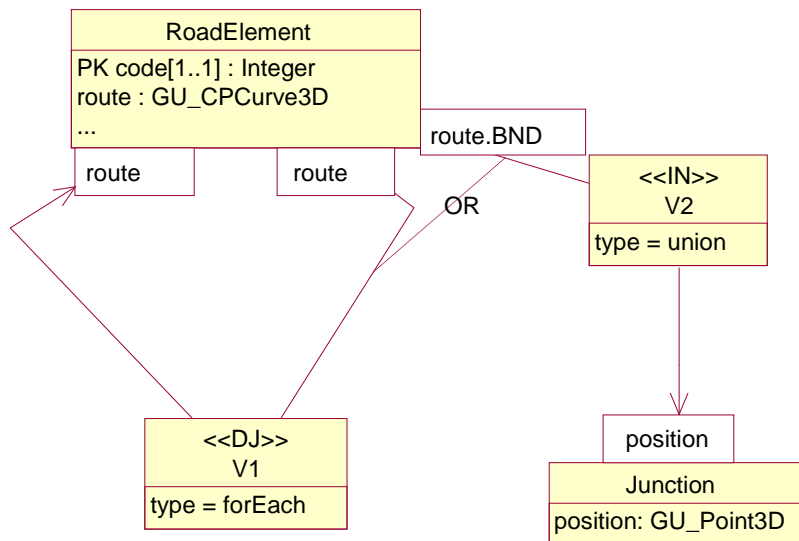


Diagram 6.6 – Example of disjunction of topological constraints

6.3 Composition constraints (part_whole constraints)

This category of constraints consists of a fundamental constraint, the composition constraint (*composedOf*) and a derivative, the partition constraint.

The partition constraint derives as it is expressible using the composition constraint and some slightly modified topological constraints. Given that the combination of modified topological constraints serves a specific function itself, an ad hoc constraint is defined, called as belonging, to express the combination.

Composition constraints are broken down as follows:

1. composition constraint, which is fundamentally non-derivable from topological constraints
2. constraints of belonging with disjunction (dj_IN e qdj_IN),
3. partition constraint, expressible through a composition constraint and a constraint of belonging with disjunction.

I should be noted that composition constraints can be involved in an expression of disjunction of constraints.

Moreover, some of the variants already introduced for topological constraints can also be used for composition constraints, with the possibility to add selections and refer the constraint to the boundary or to the planar projection of the geometric value. Finally, the constraint can also be referred to the geometry of attributes such as segmented or subregions attributes and link them to association.

6.3.1 Composition constraint

The composition constraint defines a constraint between a geometric attribute f of a class Y and the geometric attribute g of a class X . This constraint establishes that for each object y of Y , the attribute f is equal to the union of geometric attributes g of one or more objects x of X ; where the constraint is linked to an association, the constraint is more stringent, requiring that for each object y of Y , the attribute f be equal to the union of geometric attributes g of all objects of X linked to y in the association. It should be noted that in all cases, the objects x of X which help satisfy the constraint in relation to an object y of Y have a geometry in $y.g$ that is contained in (or equal to) the geometry of $y.f$.

The composition constraint is of an existential nature: indeed, it requires that, given the geometric attribute f of an object of the constrained class (composed class), there exists in the constraining class (component class) a number of instances with spatial components, that in geometric union, are equal to f .

Syntax:

constraint $Y.f$ *composedOf* $X.g$

The graphic representation of the constraint is the same as the existential constraint but using the wording “composedOf” in place of the spatial relation and the constraint type indication.

The composition constraint requires that the geometries of the constrained and constraining classes must all be of the same dimension (for example, only curves or only surfaces) as it is not possible to compose an object of a dimension different from that of the components (for example, a surface cannot be obtained through the composition of curves).

Example

The following two examples refer to the constrained class *Road* and use the composition constraint on both the areal spatial component (*relevance*) and the linear spatial component (*analyticalRoute*):

1. *Road.relevance.surface composedOf RoadArea.extension.surface*
 - this constraint requires the existence of a certain number of road areas (*RoadArea* class) that make up the *relevance* of a road;
 - the constraint indirectly requires that the extension of *RoadArea* be cut where a road ends;
 - notice that the need to refer to the *surface* attribute since both *relevance* and *extension* are surfaces with a 3D boundary.
2. *Road.analyticalRoute composedOf RoadElement.route*
 - This constraint requires the existence of a certain number of road elements (*RoadElements*) that make up the *analyticalRoute* of a road.
 - the constraint indirectly requires that the route of *RoadElements* be cut where the route of a road ends.

this constraint operates on classes and spatial components which must be defined as follows:

class *Road* (...)

attributes

class spatial components

... - *relevance*: GU_CPSurfaceB3D

... - *analyticalRoute*: GU_CPCurve3D

```
class RoadArea (...)  
  attributes  
    class spatial components  
    ... - extension: GU_CPSurfaceB3D  
class RoadElement (...)  
  attributes  
    class spatial components  
    ... - route: GU_CPCurve3D
```

6.3.2 Constraint of belonging

The constraint of belonging (geometric containment) of the spatial component g of an instance of the constrained class X to the spatial component f of an instance of the constraining class Y is obtained simply by using an existential topological constraint which requires the topological relation IN between g and f .

It would therefore not be necessary to define any specific constraint to express this property. However, it is often beneficial to indicate which topological relations are supported between the instances of the constrained class which belong to the same instance of the constraining class.

In order to satisfy this requirement, the following two constraints are introduced, which combine belonging (IN) with the disjunction of elements of belonging:

- **constraint of belonging with disjunction (*dj-IN*)**: this constraint requires that the spatial component of each instance of the constrained class (included geometry) be geometrically contained within (topological constraint IN) the spatial component of an instance of the constraining class (including geometry) and that there exists, among the included geometries contained in the same including geometry, the relation *Disjoint* or *Touches* restricted to the case where only the *boundary-boundary* intersection exists (note: the *Touches* relation also admits the *boundary-interior* intersection);
- **constraint of belonging with quasi-disjunction (*qdj-IN*)**: this constraint only applies for geometric objects of type GU_C*Curve*D (including specialisations) and allows for the existence between instances of the constrained class, in addition to *Disjoint* and *Touches* relations, of the *Cross* relation.

The graphic representation is obtained from the composition constraint by replacing the wording <<composedOf>> with <<dj-IN>> or <<qdj-IN>>.

Example

The following constraint says that the route of a natural hydrographical sub-network (*HydroSubNetwork*) must belong to a hydrographical network (*HydroNetwork*) and that all sub-networks belonging to a network must be *Disjoint* or in a *Touches* relation (*boundary-boundary*).

```
HydroSubNetwork.route dj-IN HydroNetwork.develop
```

this constraint operates on classes and spatial components which must be defined as follows:

```
class HydroSubNetwork (...)  
  attributes  
  class spatial components  
  ... - route: GU_CXCurve3D  
class HydroNetwork (...)  
  attributes  
  class spatial components  
  ... - develop: GU_CXCurve3D
```

Some of the variants already introduced for topological constraints can also be used for constraints of belonging, with the possibility to add selections and refer the constraint to the boundary, to the planar projection of the geometric value, to the geometry of segmented, events and subregions attributes or to link it to an association.

6.3.3 Partition constraint

The partition constraint expresses the combination of a composition constraint with a constraint of belonging with disjunction (in the two version *dj* and *qdj*).

The partition constraint is specified as follows:

Syntax:

```
constraint Y.f partitioned X.g  
or  
constraint Y.f q-partioned X.g
```

with the following meaning:

- The union of the geometric attributes *g* of objects from the class *X* which partition an object of the class *Y* composes the geometric attribute *f* of the object class *Y* (i.e. the constraint *Y.f composedOf X.g* applies);
- The geometric attributes *g* of the objects of class *X* which form the partition of *f* do not overlap (they are adjacent at most), so the constraint *X.g dj-IN Y.f* (or *X.g qdj-IN Y.f*) applies.

The graphic representation is taken form that of the composition constraint replacing the wording <<composedOf>> with <<partitioned>> or <<q-partitioned>>.

Example

Consider two classes representing Regions and Provinces linked by an Region-Province association, which is independent of the geometric representation of their territories. If we draw up this geometric representation in polygon form, we can see that there is a partition constraint between the polygon of a region and those of its provinces, which we can then define with the partitioned constraint as follows:

```
constraint Region.extension partitioned Region.ProvinceOfTheRegion.extension
```

this constraint operates on classes and spatial components which must be defined as follows:

```
class Region (...)  
  attributes  
    class spatial components  
    ... - extension: GU_CPSurface2D;  
  roles: ProvinceOfTheRegion [1..*]: Province  
  ...  
classe Province (...)  
  attributes  
    class spatial components  
    .... - extension: GU_CPSurface2D;
```

This example shows the combination of the partition constraint with the reference to an association, applying the general rule that the variants introduced for the existential constraint can also be applied to composition constraints.

6.3.4 Composition constraints with multiple constraining classes

There exists a variant of the *composedOf*, *partitioned* or *q-partitioned* constraints that allows them to refer to the union of values of the geometric attributes of different constraining classes. The different classes and their respective attributes must in this case be listed between parentheses, as show in the following example.

Comment and Example

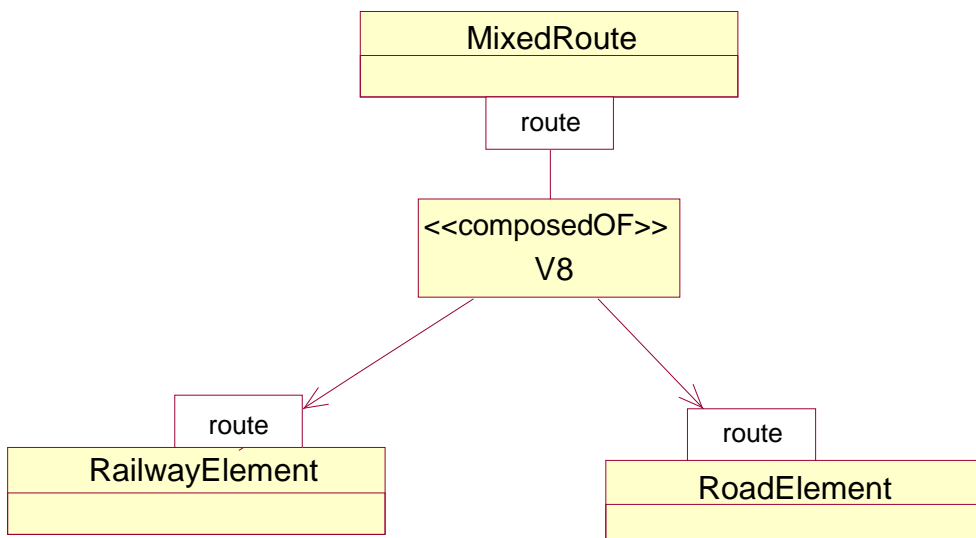
Suppose that we need to define a “mixed route” made up of road elements and railway elements; the constraint that can express this property can be specified in GeoUML only by indicating as constraining classes both the class that represents road elements and that which represents railway elements, as in the following example:

```
constraint MixedRoute.route  
  composedOf (RouteElement.route, RailwayElement.route)
```

this constraint operates on classes and spatial components which must be defined as follows:

```
class RoadElement (...)  
  attributes  
    class spatial components  
    ... - route: GU_CPCurve2D  
class RailwayElement (...)  
  attributes  
    class spatial components  
    ... - route: GU_CPCurve2D  
class MixedRoute (...)  
  attributes  
    class spatial components  
    ... - route: GU_CPCurve2D
```

The meaning of this specification is that a “mixed route” object possesses a geometric attribute made up of the union of curves belonging to the geometric attributes of road elements and railway elements. Its graphic form is shown in the following diagram, where the arrow from the constrained class is divided to point to all constraining classes.



Appendix A – Translation of constraints in OCL

A.1. Introduction

This Annex gives formal definitions in OCL of all spatial integrity constraints presented in Chapter 6.

Each constraint is indeed a constraint template in OCL, which means that the OCL formula that defines the constraint semantics in OCL contains parameters. For each constraint template a short description is given, along with the template syntax and its definition in OCL.

The templates applied to constraints used in a GeoUML schema produce the constraints formulation in OCL. These OCL formulas can be added to the application schema (AS) corresponding to the GeoUML schema in order to obtain a specification with total conformity with the rules set forth by standard ISO 19109 for the drafting of AS.

A.2. Existential topological constraint

A.2.1 Basic form

Basic existential constraint

Definition of symbols:

Given two classes *X* and *Y* each containing at least one geometric attribute, respectively *g* and *f*, the existential topological constraint from *X* to *Y* based on the disjunction of relations $DJ_R = \{rel_1, \dots, rel_n\}$ is defined as follows:

Syntax:

```
constraint X.g (rel1 | ... | reln) exists Y.f
```

OCL template:

```
ExistentialTopoConstraint (X, g, Y, f, DJ_R) ≡  
context X  
inv: Y.allInstances ->  
    exists(a:Y | self.g.check(DJ_R, a.f))
```

If one or both geometric attributes *f* and *g* are of the type `GU_C*SurfaceB3D`, it is necessary to specify the component in question, namely the attribute `B3D` or the attribute `surface`; in the syntactic definition `X.g (Y.f)` must be replaced with `X.g.B3D (Y.f.B3D)` or with `X.g.surface (Y.f.surface)`, while also replacing, where applicable, in the OCL template `self.g` with `self.g.B3D (self.g.surface)` and `a.f` with `a.f.B3D (a.f.surface)`.

A constraint can also be worked on the geometry of segmented, events or subregions attributes; in this case, the geometric attribute in the OCL template must be replaced with the call of one of the functions that return the segments, events or subregions according to the attribute type (for example, the function `SegmentsOf_A()` and `SubregionsOf_A()`, where *A* is the name of the segmented or subregions attribute).

For segmented attributes, where the geometric attribute of one or both classes involved is of the type `GU_C*SurfaceB3D`, it is necessary to specify the function that returns the segments of the 2D or 3D boundary of the surface component.

The constraint is reformulated as follows for the segmented attributes (hereinafter reference is made to the segmented attribute, however each OCL template can be reformulated in an equal manner for attributes of segmented attributes on 2D/3D boundary, events and subregions attributes).

Variant on segmented attributes (segments/segments)

Definition of symbols:

Given two classes X and Y each containing at least one segmented attribute, named a and b , respectively, the existential topological constraint (segment/segment) from X to Y , based on the disjunction of relations $DJ_R=\{rel_1,\dots,rel_n\}$ is defined as follows:

Syntax:

```
constraint X.SegmentsOf_a() (rel1|...|reln)  
           exists Y.SegmentsOf_b()
```

OCL template:

```
ExistentialTopoConstraintTR/TR (X, a, Y, b, DJ_R) ≡
```

```
context X
```

```
inv: self.SegmentsOf_a() -> forall(t:GU_Object |  
    Y.allInstances.SegmentsOf_b() ->  
    exists(a:GU_Object | t.check(DJ_R, a)))
```

Note that when the constraint refers to the geometry of a segmented attribute, the disjunction of topological relations must be satisfied by all segments returned by the function `SegmentsOf_A()`. Therefore, the result of the constraint evaluation also depends on the values returned by this function.

The segments reference may also be found on the constrained class only or on the constraining class only. For completeness, these latter two variants are also shown below.

Variant on segmented attributes (segments/geometries)

Definition of symbols:

Given two classes X and Y where X contains a segmented attribute (named a) and Y contains a geometric attribute f , the existential topological constraint from X to Y , variant segments/geometries, based on the disjunction of relations $DJ_R=\{rel_1,\dots,rel_n\}$ is defined as follows:

Syntax:

```
constraint X.SegmentsOf_a() (rel1 | ... | reln) exists Y.f
```

OCL template:

```
ExistentialTopoConstraintTR/GEO (X, a, Y, f, DJ_R) ≡
```

```
context X
```

```
inv: self.SegmentsOf_a() -> forall(t:GU_Object |  
    Y.allInstances.f ->  
    exists(a:GU_Object | t.check(DJ_R, a)))
```

Variant on segmented attributes (geometries/segments)

Definition of symbols:

Given two classes X and Y where X contains a geometric attribute g and Y contains a segmented attribute (named b), the existential topological constraint from X to Y , variant geometries/segments, based on the disjunction of relations $DJ_R=\{rel_1,\dots,rel_n\}$ is defined as follows:

Syntax:

```
constraint X.g (rel1 | ... | reln) exists Y.SegmentsOf_b()
```

OCL template:

```
ExistentialTopoConstraintGEO/TR (X, g, Y, b, DJ_R) ≡  
context X  
inv: Y.allInstances.SegmentsOf_b() ->  
exists(a:GU_Object | self.g.check(DJ_R, a))
```

A.2.2 Variant with selection

An initial variant is used to select objects from the classes involved in the constraint.

Variant with selection

Definition of symbols:

Given two classes X and Y each containing at least one geometric attribute, respectively g and f , the existential topological constraint from X to Y , variant with selection, based on the disjunction of relations $DJ_R=\{rel_1,\dots,rel_n\}$ is defined as follows:

Syntax:

```
constraint (σ1(X))X.g (rel1 | ... | reln) exists (σ2(X,Y))Y.f
```

OCL template:

```
ExistentialTopoConstraintSEL (X, σ1(X), g, Y, σ2(X,Y), f, DJ_R) ≡  
context X  
inv: σ1(self) implies  
(Y.allInstances -> exists(a:Y | σ2(self, a) and  
self.g.check(DJ_R, a.f)))
```

The selection clause $\sigma_1(X)$ is a propositional formula of the type $[not](\alpha_1 \text{ opLogico } \dots \text{ opLogico } \alpha_n)$ with $\text{opLogico} \in \{AND, OR\}$ and α_i is an atomic formula of the type: $(X.a \text{ op } X.b)$, $(X.a \text{ op } cost)$, $(X.a = null) \text{ o } (X.a = \text{not null})$ where both attributes belong to the class X and may be multi-value attributes, $\text{op} \in \{=, <>, <, >, \geq, \leq\}$ and $cost$ is a constant value other than $null$. In the first two formulae, in the presence of at least one multi-value operand ($X.a$ or $X.b$), the formula is evaluated as follows: all possible combinations (Cartesian product) between operand values are generated and the atomic formula is evaluated as “true” where there exists at least one pair of values that satisfies the comparison condition; note that in the case of single-value attributes, this corresponds to generating a single pair of values to be compared. The last two formulae verify the existence or inexistence of the null value of the attribute.

$\sigma_2(X,Y)$ is a propositional formula analogous to $\sigma_1(X)$ where α_i also accepts a formula of the join type $(Y.a \text{ op } X.b)$ provided that an attribute of the constrained class is always involved.

All attributes of constrained (constraining) classes are admitted in selection clauses, with the exception of other geometric attribute, geometric attributes of segmented (events, subregions) attributes and additional attributes of hierarchical enumerated domain; in case of DataType, the specific attribute of the DataType used in the selection clause must be specified.

The presented selection formulae allow for two cases:

1. **normal selection:** logical expressions of simple predicates of the type “value compactor attribute”; in this case are included: all possible version of $\sigma_1(X)$ and the versions of $\sigma_2(X,Y)$ where it contains only attributes of Y class.
2. **join selection:** this form accepts predicates of the type “constrainedClass.attribute comparator attribute”; in this case are included: the versions of $\sigma_2(X,Y)$ where both attributes of X and of Y are compared. Such formulae allow to link each object of the constrained class to a subset of object of the constraining class to be used for constraint satisfaction.

For selection on the constraining class $\sigma_2(X,Y)$, reference to the constraining class may be omitted, but not the reference to the constrained class; more specifically, the latter must be preceded by the symbol “\$” in the event that a constraint involves a class both as a constrained class and as a constraining class (self-ring constraint).

In addition to the selection clause applied to restrict objects involved in the constraint, it is also possible to express the selection condition to restrict any segments (events or subregions) that may be involved in a constraint by exploiting the condition of the functions `SegmentsOf_A(selection_cond)`. When appearing in a constraint, these functions can be used to specify as a parameter an enriched selection clause, with respect to what required in their definition, which is then translated in the required syntax.

The version of the template with selection extended to segmented attributes is presented below. In the template formulation we use the selection clauses $\sigma_{1a}(X, a)$ and $\sigma_{2b}(X, Y, b)$ that have to be interpreted as follows:

- $\sigma_{1a}(X, a)$ is a propositional formula of the same structure of $\sigma_1(X)$ i.e. $[not](\alpha_1 opLogico \dots opLogico \alpha_n)$ with $opLogico \in \{AND, OR\}$, but where α_i is an atomic formula that always involves the segmented attribute $X.a$ and is of the type: $(X.a op cost)$, $(X.a op X.b)$, $(X.a = null)$ or $(X.a = not null)$, where $op \in \{=, <>, <, >, \geq, \leq\}$, $cost$ is a constant value other than *null*; when the attribute $X.b$ is multi-value, the condition $(X.a op X.b)$ is transformed to a disjunction of atomic formulae $(X.a op val)$, where val corresponds in each formula to one of the values of the multi-value attribute $X.b$.
- $\sigma_{2b}(X, Y, b)$ is a propositional formula analogous to $\sigma_{1a}(X, a)$ where α_i also accepts the formula $(Y.b op X.c)$ dove $Y.b$ is the segmented attribute and $X.c$ is an attribute of the constrained class.

Regarding the attributes admitted both selection clauses use all class attributes with the exceptions described above for objects selection conditions.

Shown below are the three versions of the constraint with selection in the cases: segments on segments, geometries on segments and segments on geometries.

Variant with selection and segmented attributes

Definition of symbols:

Given two classes X and Y each containing at least one geometric attribute, respectively g and f , and a segmented attribute, respectively a and b , the existential topological constraint from X to Y , variant with selection and segmented attributes, based on the disjunction of relations $DJ_R = \{rel_1, \dots, rel_n\}$ is defined as follows:

segments/segments

Syntax:

```
constraint ( $\sigma_1(X)$ )X.SegmentsOf_a( $\sigma_{1a}(X, a)$ )
              ( $rel_1, \dots, rel_n$ ) exists
              ( $\sigma_2(X, Y)$ )Y.SegmentsOf_b( $\sigma_{2b}(X, Y, b)$ )
```

OCL template:

```
ExistentialTopoConstraintTR/TRSEL
  (X,  $\sigma_1(X)$ , a,  $\sigma_{1a}(X, a)$ , Y,  $\sigma_2(X, Y)$ , b,  $\sigma_{2b}(X, Y, b)$ , DJ_R)  $\equiv$ 
```

context X

```
inv:  $\sigma_1(\text{self})$  implies
      self.SegmentsOf_a(" $\sigma_{1a}(X, a)$ ") ->
        forall(t:GU_Object |
              Y.allInstances->
                select(y:Y |  $\sigma_2(\text{self}, y)$ ).SegmentsOf_b(" $\sigma_{2b}(X, Y, b)$ ") ->
                  exists(a:GU_Object | t.check(DJ_R, a))
              )
```

geometries/segments

Syntax:

```
constraint ( $\sigma_1(X)$ )X.g
              ( $rel_1, \dots, rel_n$ ) exists
              ( $\sigma_2(X, Y)$ )Y.SegmentsOf_b( $\sigma_{2b}(X, Y, b)$ )
```

OCL template:

```
ExistentialTopoConstraintGEO/TRSEL
  (X,  $\sigma_1(X)$ , g, Y,  $\sigma_2(X, Y)$ , b,  $\sigma_{2b}(X, b)$ , DJ_R)  $\equiv$ 
```

context X

```
inv:  $\sigma_1(\text{self})$  implies
      Y.allInstances->
        select(y:Y |  $\sigma_2(\text{self}, y)$ ).SegmentsOf_b(" $\sigma_2(X, Y, b)$ ") ->
          exists(a:GU_Object | self.g.check(DJ_R, a))
```

segments/geometries

Syntax:

```
constraint ( $\sigma_1(X)$ )X.SegmentsOf_a( $\sigma_{1a}(X,a)$ )
( $rel_1, \dots, rel_n$ ) exists
( $\sigma_2(X,Y)$ )Y.f
```

OCL template:

```
ExistentialTopoConstraintTR/GEOSEL
(X,  $\sigma_1(X)$ , a,  $\sigma_{1a}(X,a)$ , Y,  $\sigma_2(X,Y)$ , f, DJ_R)  $\equiv$ 
```

context X

```
inv:  $\sigma_1(\text{self})$  implies
self.SegmentsOf_a(" $\sigma_1(X,a)$ ") ->
forall(t:GU_Object |
Y.allInstances->exists(a:Y |  $\sigma_1(\text{self},a)$ ) and
t.check(DJ_R, a))
)
```

A.2.3 Variant on the boundary or planar projection

Variant on boundary and planar projection

Definition of symbols:

Given two classes X and Y each containing at least one geometric attribute, respectively g and f, the existential topological constraint from X to Y, variant on the boundary, based on the disjunction of relations $DJ_R=\{rel_1,\dots,rel_n\}$ is defined as follows:

Syntax:

```
constraint X.g.BND ( $rel_1$  | ... |  $rel_n$ ) exists Y.f
```

OCL template:

```
ExistentialTopoConstraintB/- (X, g, Y, f, DJ_R)  $\equiv$ 
```

context X

```
inv: Y.allInstances ->
exists(a:Y | self.g.boundary().check(DJ_R, a.f))
```

the existential topological constraint from X to Y, variant on the planar projection, based on the disjunction of relations $DJ_R=\{rel_1,\dots,rel_n\}$ is also defined as follows:

Syntax:

```
constraint X.g.PLN ( $rel_1$  | ... |  $rel_n$ ) exists Y.f
```

OCL template:

```
ExistentialTopoConstraintP/-
(X, g: GU_Object, Y, f: GU_Object, DJ_R)  $\equiv$ 
```

context X

```
inv: Y.allInstances ->
exists(a:Y | self.g.planar().check(DJ_R, a.f))
```

Similarly the variants on boundary or on planar projection applied to the attribute f of the constraining class can be defined:

ExistentialTopoConstraint^{-/B},
ExistentialTopoConstraint^{-/P}

or the variants in which applies boundary on constrained class and planar projection on the constraining class or all other combinations:

ExistentialTopoConstraint^{B/B},
ExistentialTopoConstraint^{P/P},
ExistentialTopoConstraint^{B/P},
ExistentialTopoConstraint^{P/B}

As well as all variants obtained from the application of the two functions in cascade.

It should be noted that the function planar() applied to a 2D geometry does not modify the geometry and that, when applied to an 3D object, it transforms it to an object in 2D space.

This variant may be combined with others which accept selection, segmented attributes or a component of a surface with 3D boundary.

A.2.4 Variant linked to an association

This variant considers, in order to satisfy the constraint, only objects of the constraining class which are linked to the object to be verified through an association specified in the GeoUML schema.

Variant linked to an association

Definition of symbols:

Given two classes X and Y each containing at least one geometric attribute, respectively g and f, and among these there exists an association, where the role of Y is r. The existential topological constraint from X to Y, variant linked to an association, based on the disjunction of relations DJ_R={rel₁,...,rel_n} is defined as follows:

Syntax:

constraint X.g (rel₁ | ... | rel_n) exists X.r.f

OCL template:

ExistentialTopoConstraint^A(X, g, r, Y, f, DJ_R) ≡

context X

inv: self.r -> exists(a:Y | self.g.check(DJ_R, a.f))

A.3 Union topological constraint

A.3.1 Basic form

Basic form

Definition of symbols:

Given two classes X and Y each containing at least one geometric attribute, respectively g and f , the union topological constraint from X to Y based on the disjunction of relations $DJ_R=\{rel_1,\dots,rel_n\}$ is defined as follows:

Syntax:

constraint X.g (rel₁ | ... | rel_n) unione Y.f

OCL template:

UnionTopoConstraint (X, g, Y, f, DJ_R) ≡

context X

inv: self.g.check(DJ_R, Y.allInstances ->
iterate(a:Y, acc: GU_Object = ∅ |
acc.gUnion(a.f))

A.3.2 Variant with selection

For the union topological constraint there are also variants presented for the existential topological constraint, that is: version with selection, version referring to boundary and planar projection, and version linked to an association.

Variant with selection

Definition of symbols:

Given two classes X and Y each containing at least one geometric attribute, respectively g and f , the union topological constraint from X to Y , variant with selection, based on the disjunction of relations $DJ_R=\{rel_1,\dots,rel_n\}$ is defined as follows:

Syntax:

constraint (σ₁(X))X.g (rel₁ | ... | rel_n) unione (σ₂(X,Y))Y.f

OCL template:

UnionTopoConstraint^{SEL}(X, σ₁(X), g, Y, σ₂(X,Y), f, DJ_R) ≡

context X

inv: σ₁(self) implies
self.g.check(DJ_R,
Y.allInstances->select(a:Y | σ₂(self,a))->
iterate(b:Y, acc: GU_Object = ∅ |
acc.gUnion(b.f))

The union constraint can also be applied to segmented attributes, considering the segmented attribute for the constrained class only, since for the constraining class this constraint requires the union of geometries, thus starting from segments does not change the geometry considered

for the constraining object. Variants with selection are shown directly, while variants without selection are obtained by simply removing the selection condition.

A.3.3 Variant with selection and segmented attributes

Variant with selection and segmented attributes

(segment/segment, geometries/segments, segments/geometries)

Definition of symbols:

Given two classes X and Y each containing at least one geometric attribute, respectively g and f , and each with one segmented attribute, respectively a and b , the union topological constraint from X to Y , variant with selection and segmented attributes, based on the disjunction of relations $DJ_R=\{rel_1,\dots,rel_n\}$ is defined as follows:

segments/geometries

Syntax:

constraint $(\sigma_1(X))X.SegmentsOf_a(\sigma_1(X,a))$
 $(rel_1 \mid \dots \mid rel_n) \underline{unione} (\sigma_2(X,Y))Y.f$

OCL template:

```
UnionTopoConstraintSELTR/GEO
(X,  $\sigma_1(X)$ , a,  $\sigma_1(X,a)$ , Y,  $\sigma_2(X,Y)$ , f, DJ_R)  $\equiv$ 
context X
inv:  $\sigma_1(self)$  implies
self.SegmentsOf_a(" $\sigma_1(X,a)$ ") -> forall(t:GU_Object |
t.check(DJ_R,
Y.allInstances-> select(a:Y |  $\sigma_2(self,a)$ ).f->
iterate(b:GU_Object,
acc: GU_Object =  $\emptyset$  |
acc.gUnion(b))))
```

geometries/segments

Syntax:

constraint $(\sigma_1(X))X.g (rel_1 \mid \dots \mid rel_n)$
unione $(\sigma_2(X,Y))Y.SegmentsOf_b(\sigma_{2b}(X,Y,b))$

OCL template:

```
UnionTopoConstraintSELGEO/TR
(X,  $\sigma_1(X)$ , g, Y,  $\sigma_2(X,Y)$ , b,  $\sigma_{2b}(X,Y,b)$ , DJ_R)  $\equiv$ 
context X
inv:  $\sigma_1(self)$  implies
self.g.check(DJ_R, Y.allInstances->
select(a:Y |  $\sigma_2(self,a)$ ).SegmentsOf_b(" $\sigma_2(X,Y,b)$ ") ->
iterate(b:GU_Object,
acc: GU_Object =  $\emptyset$  |
acc.gUnion(b))))
```

segments/segments

Syntax:

constraint ($\sigma_1(X)$)X.SegmentsOf_a($\sigma_1(X,a)$) (rel₁ | ... | rel_n)
unione ($\sigma_2(X,Y)$)Y.SegmentsOf_b($\sigma_{2b}(X,Y,b)$)

OCL template:

UnionTopoConstraint^{SEL_{TR/TR}}
(X, $\sigma_1(X)$, a, $\sigma_1(X,a)$, Y, $\sigma_2(X,Y)$, b, $\sigma_{2b}(X,Y,b)$, DJ_R)≡

context X

inv: $\sigma_1(\text{self})$ implies

```
self.SegmentsOf_a("σ1(X,a)")-> forall(t:GU_Object |
  t.check(DJ_R, Y.allInstances-> select(a:Y |
    σ2(self,a)).SegmentsOf_b("σ2b(X,Y,b)")->
    iterate(b:GU_Object,
      acc: GU_Object = ∅ |
      acc.gUnion(b))))
```

A.4. Universal topological constraint

A.4.1 Basic form

Basic form

Definition of symbols:

Given two classes X and Y each containing at least one geometric attribute, respectively g and f , the universal topological constraint from X to Y based on the disjunction of relations $DJ_R=\{rel_1,\dots,rel_n\}$ is defined as follows:

Syntax:

constraint X.g (rel₁ | ... | rel_n) forEach Y.f

OCL template:

UniversalTopoConstraint(X, g, Y, f, DJ_R) ≡

context X

inv: Y.allInstances->forEach(a:Y| self.g.check(DJ_R, a.f))

A.4.2 Variant with selection

Variant with selection

Definition of symbols:

Given two classes X and Y each containing at least one geometric attribute, respectively g and f , the universal topological constraint from X to Y , variant with selection, based on the disjunction of relations $DJ_R=\{rel_1,\dots,rel_n\}$ is defined as follows:

Syntax:

constraint ($\sigma_1(X)$)X.g (rel₁ | ... | rel_n) forEach ($\sigma_2(X,Y)$)Y.f

OCL template:

UniversalTopoConstraint^{SEL}
(X, $\sigma_1(X)$, g, Y, $\sigma_2(X,Y)$, f, DJ_R) ≡

context X

inv: $\sigma_1(\text{self})$ implies
Y.allInstances -> select(a:Y| $\sigma_2(\text{self},a)$ ->
forEach(b:Y| self.g.check(DJ_R, b.f))

A.4.3 Variant with selection and segmented attributes

It is also possible to apply the universal constraint on segmented, events, or subregions attributes with the following semantics.

Variant with selection and segmented attributes

(segment/segment, geometries/segments, and segments/geometries)

Definition of symbols:

Given two classes X and Y each containing at least one geometric attribute, respectively g and f, the universal topological constraint from X to Y, variant with selection and segmented attributes, based on the disjunction of relations DJ_R={rel₁,...,rel_n} is defined as follows:

segments/segments

Syntax:

```
constraint (σ1(X))X.aTratti_a(σ1a(X,a))
              (rel1 | ... | reln) forEach
              (σ2(X,Y))Y.aTratti_b(σ2b(X,Y,b))
```

OCL template:

```
UniversalTopoConstraintSELTR/TR
  (X, σ1(X), a, σ1a(X,a), Y, σ2(X,Y), b, σ2b(X,Y,b), DJ_R) ≡
context X
inv: σ1(self) implies
self.SegmentsOf_a("σ1a(X,a)")->
  forall(t:GU_Object |
    Y.allInstances->
      select(y:Y| σ2(self,Y)).SegmentsOf_b("σ2b(X,Y,b)")->
        forEach(c:GU_object| t.check(DJ_R, c))
  )
```

geometries/segments

Syntax:

```
constraint (σ1(X))X.g
              (rel1 | ... | reln) forEach
              (σ2(X,Y))Y.aTratti_b(σ2b(X,Y,b))
```

OCL template:

```
UniversalTopoConstraintSELGEO/TR
  (X, σ1(X), g, Y, σ2(X,Y), b, σ2b(X,Y,b), DJ_R) ≡
context X
inv: σ1(self) implies
  Y.allInstances->
    select(y:Y| σ2(self,Y)).SegmentsOf_b("σ2b(X,Y,b)")->
      forEach(c:GU_object | self.g.check(DJ_R, c))
```

segments/geometries

Syntax:

```
constraint ( $\sigma_1(X)$ )X.aTratti_a( $\sigma_{1a}(X,a)$ )  
  ( $rel_1 \mid \dots \mid rel_n$ ) forEach  
  ( $\sigma_2(X,Y)$ )Y.f
```

OCL template:

```
UniversalTopoConstraintSELTR/GEO  
  (X,  $\sigma_1(X)$ , a,  $\sigma_{1a}(a,X)$ , Y,  $\sigma_2(X,Y)$ , f, DJ_R)  $\equiv$   
context X  
inv:  $\sigma_1(\text{self})$  implies  
self.SegmentsOf_a(" $\sigma_{1a}(X,a)$ ") ->  
  forall(t:GU_Object |  
    Y.allInstances->select(b:Y |  $\sigma_2(\text{self},b)$ )->  
      forEach(c:Y | t.check(DJ_R, c.f))  
  )
```

A.5. Composition constraint

A.5.1 Basic form

Basic form

Definition of symbols:

Given a class Y with a geometric attribute f and a class X with a geometric attribute g , the composition constraint from Y to X is defined as follows:

Syntax:

constraint $Y.f$ compostoDa $X.g$

OCL template:

ComposedOfConstraint(Y, f, X, g) \equiv

context Y

inv: self.f.Equals($X.allInstances.g$ ->
select($a:GU_Object$ | self.f.Contains(a) or
self.f.Equals(a)) ->
iterate($b:GU_Object, acc: GU_Object = \emptyset$ |
acc.gUnion(b)))

If one or both geometric attributes f and g are of type $GU_C*SurfaceB3D$ it is necessary to specify the component in question, namely the attribute $B3D$ or *surface*; accordingly, in the syntactic definition of the constraint in the case of both attributes, $X.g$ must be replaced with $X.g.B3D$ (*surface*) and $Y.f$ with $Y.f.B3D$ (*surface*) while also replacing, where applicable, in the OCL template, self.f with self.f.B3D (*surface*), $X.AllInstances.g$ with $X.AllInstances.g.B3D$ (*surface*) and self.f.Contains(a) with self.f.B3D(*surface*).Contains(a).

A.5.2 Variant with selection

Shown below is the definition of the composition constraint with selection illustrating how the other constraints are altered in the presence of selection.

Variant with selection

Definition of symbols:

Given a class Y with a geometric attribute f and a class X with a geometric attribute g , the composition constraint from Y to X , variant with selection, is defined as follows:

Syntax:

constraint ($\sigma_1(Y)$) $Y.f$ compostoDa ($\sigma_2(Y,X)$) $X.g$

OCL template:

ComposedOfConstraint^{SEL}($Y, \sigma_1(Y), f, X, \sigma_2(Y,X), g$) \equiv

context Y

inv: σ_1 (self) implies (self.f.Equals(
 $X.allInstances$ ->select($x:X$ | σ_2 (self, x)). g ->
select($a:GU_Object$ | self.f.Contains(a)
or self.f.Equals(a))->
iterate($b:GU_Object, acc: GU_Object = \emptyset$ |
acc.gUnion(b)))

A.5.3 Variant with selection and segmented attributes

The constraint is reformulated as follows for segmented attributes:

Variant with selection and segmented attribute

(segments/segments, geometries/segments and segments/geometries)

Definition of symbols:

Given a class Y with a geometric attribute f and a segmented attribute b , and a class X with a geometric attribute g and a segmented attribute a , the composition constraint from Y to X , variant with selection and segmented attributes, is defined as follows:

segments/segments

Syntax:

constraint $(\sigma_1(Y))Y.$ SegmentsOf_b $(\sigma_{1b}(Y,b))$ compostoDa
 $(\sigma_2(Y,X))X.$ SegmentsOf_a $(\sigma_{2a}(Y,X,a))$

OCL template:

ComposedOfConstraint^{SEL}_{TR/TR}($Y, \sigma_1(Y), b, \sigma_{1b}(Y,b),$
 $X, \sigma_2(Y,X), a, \sigma_{2a}(Y,X,a)$) \equiv

context Y

inv: $\sigma_1(\text{self})$ implies

```
self.SegmentsOf_b("σ1b(Y,b)") ->
forall(t: GU_Object | t.Equals(
  X.allInstances->
    select(x:X | σ2(self,x)).SegmentsOf_a("σ2a(Y,X,a)") ->
    select(c:GU_Object | t.Contains(c) or t.Equals(c)) ->
      iterate(d:GU_Object, acc: GU_Object = ∅ |
        acc.gUnion(d)
      )
  )
)
```

geometries/segments

Syntax:

constraint $(\sigma_1(Y))Y.f$ compostoDa
 $(\sigma_2(Y,X))X.$ SegmentsOf_a $(\sigma_{2a}(Y,X,a))$

OCL template:

ComposedOfConstraint^{SEL}_{GEO/TR}($Y, \sigma_1(Y), f, X, \sigma_2(Y,X),$
 $a, \sigma_{2a}(Y,X,a)$) \equiv

context Y

inv: $\sigma_1(\text{self})$ implies

```
self.f.Equals(
  X.allInstances->
    select(x:X | σ2(self,x)).SegmentsOf_a("σ2a(Y,X,a)") ->
    select(c:GU_Object | self.f.Contains(c)
      or self.f.Equals(c)) ->
      iterate(d:GU_Object, acc: GU_Object = ∅ |
        acc.gUnion(d)
      )
  )
)
```

segments/geometries

Syntax:

constraint $(\sigma_1(Y))Y.SegmentsOf_b(\sigma_{1b}(Y,b))$ compostoDa
 $(\sigma_2(Y,X))X.g$

OCL template:

```
ComposedOfConstraintSELTR/GEO(Y,  $\sigma_1(Y)$ , b,  $\sigma_{1b}(Y,b)$ ,
X,  $\sigma_2(Y,X)$ , g)  $\equiv$ 
context Y
inv:  $\sigma_1(self)$  implies
self.SegmentsOf_b(" $\sigma_{1b}(Y,b)$ ") ->
  forall(t: GU_Object | t.Equals(
    X.allInstances ->
      select(c:X |  $\sigma_2(self,c)$ 
        and (t.Contains(c.g) or t.Equals(c.g)).g) ->
        iterate(d:GU_Object, acc: GU_Object =  $\emptyset$  |
          acc.gUnion(d)))
  )
```

A.5.4 Variant on boundary and planar projection

Below is the formal definition for the composition constraints on boundary and planar projection.

Variant on boundary

Definition of symbols:

Given a class *Y* with a geometric attribute *f* and a class *X* with a geometric attribute *g*, the composition constraint from *Y* to *X*, variant on boundary, is defined as follows:

Syntax:

constraint $Y.f.BND$ compostoDa $X.g$

OCL template:

```
ComposedOfConstraintB-(Y, f, X, g)  $\equiv$ 
context Y
inv: self.f.boundary().Equals(
  X.allInstances.g->
    select(a:GU_Object | self.f.boundary().Contains(a)
      or self.f.boundary().Equals(a)) ->
    iterate(b:GU_Object, acc: GU_Object =  $\emptyset$  |
      acc.gUnion(b))
```

Similarly the variants on planar projection applied to the attribute *f* of *Y* can be defined:

ComposedOfConstraint^{P⁻}

or the variants in which applies boundary on constrained class and planar projection on the constraining class or all other combinations:

ComposedOfConstraint^{-/B},
ComposedOfConstraint^{-/P},
ComposedOfConstraint^{B/B},


```
ComposedOfConstraintP/P  
ComposedOfConstraintB/P  
ComposedOfConstraintP/B
```

As well as all variants obtained from the application of the two functions in cascade.

A.5.5 Variant linked to an association

Finally, the formal definition is given for the composition constraint based on associations, in which composition constraint must not refer to two independent class and to the property of geometric containment, but must be based on an association that connects them.

This means that the objects of the constraining class that must participate in the constraint are those which participate in the association. This variant of the composition constraint is defined below:

Variant linked to an association

Definition of symbols:

Given a class Y with a geometric attribute f and a role r towards a class X with a geometric attribute g, the composition constraint from Y to X, variant linked to association, is defined as follows:

Syntax:

```
constraint Y.f compostoDa Y.r.g
```

OCL template:

```
ComposedOfOnAssociation(Y, f, r, g) ≡
```

```
context Y
```

```
inv: self.f.Equals(self.r.g ->  
    iterate(b:GU_Object, acc: GU_Object = ∅ |  
        acc.gUnion(b)))
```

As we can see, the only difference with respect to the constraint not linked to an association is the fact that, instead of being based on all objects of X (`X.allInstances`), it is based solely on instances attainable through the role r of the association (`self.r`).

A.6. Constraint of belonging

A.6.1 Basic form

Constraint of belonging with disjunction

Definition of symbols:

Given a class *X* with a geometric attribute *g* and a class *Y* with a geometric attribute *f*, the constraint of belonging with disjunction from *X* to *Y* is defined as follows:

Syntax:

```
constraint X.g dj-IN Y.f
```

OCL template:

```
dj-IN(X, g, Y, f) ≡
```

```
context X
```

```
inv: Y.allInstances.f->
```

```
exists(a:GU_Object|self.g.In(a)  
or self.g.Equals(a))
```

```
and
```

```
X.allInstances ->
```

```
select(x | (x.OID ≠ self.OID)).g->
```

```
forall(v: GU_Object |brotherIN(v, self.g, Y, f)
```

```
implies
```

```
(v.Disjoint(self.g) or
```

```
v.Touch(self.g) and v.Touch(self.g.boundary()))))
```

```
brotherIN(x1, x2, Y, f)
```

```
≡ Y.allInstances ->
```

```
exists(y:Y| ((x1.In(y.f)) or (x1.Equals(y.f))
```

```
and
```

```
((x2.In(y.f) or (x2.Equals(y.f)))
```

Constraint of belonging with quasi-disjunction

Definition of symbols:

Given a class *X* with a geometric attribute *g* and a class *Y* with a geometric attribute *f*, the constraint of belonging with quasi-disjunction from *X* to *Y* is defined as follows:

Syntax:

`constraint X.g qdj-IN Y.f`

OCL template:

`qdj-IN (X, g, Y, f) ≡`

context X

inv: Y.allInstances.f->

exists(a: GU_Object | self.g.In(a) or
self.g.Equals(a))

and

X.allInstances ->

select(x | (x.OID ≠ self.OID).g->

forall(v: GU_Object | brotherIN(v, self.g, Y, f)

implies

(v.Disjoint(self.g) or

v.Touch(self.g) or

v.Cross(self.g)))

where brotherIN() is the function defined in the basic form.

Some of the variants already introduced for topological constraints can also be used for constraints of belonging.

A.6.2 Variant with selection

Variant with selection

Definition of symbols:

Given a class *X* with a geometric attribute *g* and a class *Y* with a geometric attribute *f*, the constraint of belonging with disjunction from *X* to *Y*, variant with selection, is defined as follows:

Syntax:

`constraint (σ1(X))X.g dj-IN (σ2(X,Y))Y.f`

OCL template:

`dj-IN(X, σ1(X), g, Y, σ2(X,Y), f) ≡`

context X

inv: σ₁(self) implies (

(Y.allInstances-> select(y:Y | σ₂(self,y)).f->

exists(a:GU_Object | self.g.In(a) or
self.g.Equals(a)))

and

(X.allInstances->

select(x:X | σ₁(x) and (x.OID ≠ self.OID)).g ->

forall(v: GU_Object | brotherIN(v, self.g, Y, f)

implies

(v.Disjoint(self.g) or

v.Touch(self.g) and v.Touch(self.g.boundary()))))

where brotherIN() is the function defined in the basic form.

This variant is trivially propagated also to the *qdj-IN* constraint.

A.6.3 Variant with selection and segmented attributes

Variant with selection and segmented attributes

(segments/segments, geometries/segments and segments/geometries)

Definition of symbols:

Given two classes *X* and *Y* each containing at least one geometric attribute, respectively *g* and *f*, and each with one segmented attribute, respectively *a* and *b*, the constraint of belonging with disjunction from *X* to *Y*, variant with selection and segmented attributes, based on the disjunction of relations $DJ_R=\{rel_1,\dots,rel_n\}$ is defined as follows:

segments/segments

Syntax:

$$\frac{\text{constraint } \sigma_1(X) X.\text{SegmentsOf_a}(\sigma_{1a}(X, a))}{\sigma_2(X, Y) Y.\text{SegmentsOf_b}(\sigma_{2b}(X, Y, b))} \text{ dj-IN}$$

OCL template:

$$dj-IN_{TR/TR}^{SEL}(X, \sigma_1(X), a, \sigma_{1a}(X, a), Y, \sigma_2(X, Y), b, \sigma_{2b}(X, Y, b)) \equiv$$

context X

inv: $\sigma_1(\text{self})$ implies

```
(self.SegmentsOf_a("σ1a(X, a)") ->
  forall(t: GU_Object |
    Y.allInstances->
      select(y:Y | σ2(self, y)).SegmentsOf_b("σ2b(X, Y, b)") ->
        exists(a:GU_Object | t.In(a) or t.Equals(a))
    and
    X.allInstances->
      select(x:X | σ1(x)).SegmentsOf_a("σ1a(X, a)") ->
        select (v: GU_Object | not (v.sameObj(t))
          or (v.sameObj(t) and
            v.sameValueSeg(t)) ->
          forall(c: GU_Object |
            brotherIN(c, t, Y, f)
            implies
            (c.Disjoint(t) or
              c.Touch(t) and c.Touch(t.boundary()))))
```

Where the function `brotherIN()` is the one defined in the basic form; `sameObj()` verifies whether the segments compared belong to the same object and finally `sameValueSeg()` verifies whether the two segments are associated with the same segmented attribute value.

geometries/segments

Syntax:

constraint $(\sigma_1(X))X.g$ *dj-IN*
 $(\sigma_2(X,Y))Y.SegmentsOf_b(\sigma_{2b}(X,Y,b))$

OCL template:

$dj-IN_{GEO/TR}^{SEL}(X, \sigma_1(X), g, Y, \sigma_2(X,Y), b, \sigma_{2b}(X,Y,b)) \equiv$

context X

inv: $\sigma_1(self)$ implies (
Y.allInstances->
 select(y:Y | $\sigma_2(self,y)$).SegmentsOf_b(" $\sigma_{2b}(X,Y,b)$ ")->
 exists(a:GU_Object | self.In(a) or
 self.Equals(a))

and

X.allInstances->
 select(x:X | $\sigma_1(x)$ and $x.OID \neq self.OID$).g->
 forall(c: GU_Object | brotherIN(c, self.g, Y, f)
 implies
 (c.Disjoint(self.g) or
 c.Touch(self.g) and c.Touch(self.g.boundary()))))

segments/geometries

Syntax:

constraint $(\sigma_1(X))X.SegmentsOf_a(\sigma_{1a}(X,a))$ *dj-IN*
 $(\sigma_2(X,Y))Y.f$

OCL template:

$dj-IN_{TR/GEO}^{SEL}(X, \sigma_1(X), a, \sigma_{1a}(X,a), Y, \sigma_2(X,Y), f) \equiv$

context X

inv: $\sigma_1(self)$ implies
(self.SegmentsOf_a(" $\sigma_{1a}(X,a)$ ")->
 forall(t: GU_Object |
 Y.allInstances->select(y:Y | $\sigma_2(self,y)$).f->
 exists(a:GU_Object | t.In(a) or t.Equals(a))
 and
 X.allInstances->
 select(x:X | $\sigma_1(x)$).SegmentsOf_a(" $\sigma_{1a}(X,a)$ ")->
 select (v: GU_Object | not (v.sameObj(t))
 or (v.sameObj(t) and
 v.sameValueTratto(t)))->
 forall(c: GU_Object |
 brotherIN(c, t, Y, f)
 implies
 (c.Disjoint(t) or
 c.Touch(t) and c.Touch(t.boundary()))))

Where the function brotherIN() is the one defined in the basic form; sameObj() verifies whether the segments compared belong to the same object and finally sameValueSeg() verifies whether the two segments are associated with the same segmented attribute value.

A.6.4 Variant on boundary and planar projection

Constraint of belonging on boundary and planar projection

Definition of symbols:

Given a class X with a geometric attribute g and a class Y with a geometric attribute f , the constraint of belonging with disjunction from X to Y , variant on boundary or planar projection, is defined as follows:

Syntax:

constraint $X.g.BND$ dj-IN $Y.f$

OCL template:

$dj-IN^{B-}(X, g, Y, f) \equiv$

context X

inv: $Y.allInstances.f \rightarrow$

$exists(a:GU_Object | self.g.boundary().In(a)$
 $or self.g.boundary().Equals(a))$

$and X.allInstances \rightarrow$

$select (x | (x.OID \neq self.OID)).g.boundary \rightarrow$

$forall(c: GU_Object |$

$brotherIN(c, self.g.boundary(), Y, f)$

$implies$

$(c.Disjoint(self.g.boundary()) or$

$c.Touch(self.g.boundary()))$

Syntax:

constraint $X.g$ dj-IN $Y.f.PLN$

OCL template:

$dj-IN^P(X, g, Y, f) \equiv$

context X

inv: $Y.allInstances.f.planar() \rightarrow$

$exists(a:GU_Object | self.g.In(a) or$
 $self.g.Equals(a))$

and

$X.allInstances.g \rightarrow$

$select(x:X | x.OID \neq self.OID).g \rightarrow$

$forall(v: GU_Object | brotherIN(v, self.g, Y, f)$

$implies$

$(v.Disjoint(self.g) or$

$v.Touch(self.g) and v.Touch(self.g.boundary()))$

Where the function `brotherIN()` is the one defined in the basic form.

A.7. Partition constraint

The partition constraint is obtained by combining a composition constraint with a constraint of belonging with disjunction or quasi-disjunction, as defined below.

Partition constraint

Definition of symbols:

Given a class *Y* with a geometric attribute *f* and a class *X* with a geometric attribute *g*, the partitioned constraint from *Y* to *X* is defined as follows:

Syntax:

constraint *Y.f* partitioned *X.g*

OCL template:

```
partitioned(Y, f, X, g) ≡  
  dj-IN(X, g, Y, f)  
  and  
  ComposedOfConstraint(Y, f, X, g)
```

Quasi-partition constraint

Definition of symbols:

Given a class *Y* with a geometric attribute *f* and a class *X* with a geometric attribute *g*, the quasi-partitioned constraint from *Y* to *X* is defined as follows:

Syntax:

constraint *Y.f* q-partitioned *X.g*

OCL template:

```
q-partitioned(X, g, Y, f) ≡  
  qdj-IN(X, g, Y, f)  
  and  
  ComposedOfConstraint(Y, f, X, g)
```

Some of the variants already introduced for topological constraints can also be used for partition constraints, with the possibility to add selections and refer the constraint to the boundary or to the planar projection of the geometric value. Finally, the constraint can also be referred to the geometry of segmented, events or subregions attributes or link it to an association.

A.8. Composition constraints with multiple constraining classes

In order to formally define the semantics of this type of constraint, it is necessary to specify new OCL templates as follows (the union function refers here to the union of objects of different geometric types, while gUnion refers to the union of point sets form geometries of geometric objects).

A.8.1 Basic form

Basic form

Definition of symbols:

Given a class Y with a geometric attribute f and a set of classes X₁, ..., X_n with a geometric attribute g₁, ..., g_n, the composition constraint from Y to X₁, ..., X_n is defined as follows:

Syntax:

constraint Y.f compostoDa (X₁.g₁, ..., X_n.g_n)

OCL template:

ComposedOfConstraint^{MULTI}(Y, f, X₁, g₁, ..., X_n, g_n) ≡

context Y

inv: self.f.Equals

```
( X1.allInstances.g1 ->
    union(X2.allInstances.g2 ->
        ...
        union(Xn.allInstances.gn)...) ->
    select(a:GU_Object | self.f.Contains(a)
        or self.f.Equals(a)) ->
    iterate(b:GU_Object, acc: GU_Object = ∅ |
        acc.gUnion(b)
    )
)
```

Also for these constraints, the syntactic definition and the OCL constraint template are modified where the geometric attributes *f*, *g₁*, ..., *g_n* are of type GU_C*SurfaceB3D following the approach described in similar previous cases.

Variants linked to an association and all other combinations are obtainable through the natural combination of the OCL expressions presented.

A.8.2 Variant with selection

Variant with selection

Definition of symbols:

Given a class Y with a geometric attribute f and a set of classes X_1, \dots, X_n with a geometric attribute g_1, \dots, g_n , the composition constraint from Y to X_1, \dots, X_n , variant with selection, is defined as follows:

Syntax:

constraint $(\sigma_1(Y))Y.f$ compostoDa
 $((\sigma_{2,1}(Y, X_1))X_1.g_1, \dots, (\sigma_{2,n}(Y, X_n))X_n.g_n)$

OCL template:

ComposedOfConstraintMulti^{SEL}($Y, f, X_1, g_1, \dots, X_n, g_n$) \equiv

context Y

inv: $\sigma_0(\text{self})$ implies

```

self.f.Equals(
  X1.allInstances->select(x1:X1 |  $\sigma_{2,1}(Y, X_1)$ ).g1 ->
  union(X2.allInstances->select(x2:X2 |
     $\sigma_{2,2}(Y, X_2)$ ).g2 ->
  ...
  union(Xn.allInstances->select(xn:Xn |
     $\sigma_{2,n}(Y, X_n)$ ).gn)) ->
select(a:GU_Object |
  self.f.boundary().Contains(a)
  or self.f.boundary().Equals(a)) ->
iterate(b:GU_Object, acc: GU_Object =  $\emptyset$  |
  acc.gUnion(b))

```

A.8.3 Variant with selection and segmented attributes

Variant with selection and segmented attributes

(segments/segments, geometries/segments and segments/geometries)

Definition of symbols:

Given a class Y with a geometric attribute f and a segmented attribute, named a , and a set of classes X_1, \dots, X_n each with a geometric attribute g_1, \dots, g_n , and a segmented attribute, named b_i , the composition constraint from Y to X_1, \dots, X_n , variant with selection and segmented attributes, is defined as follows:

segments/segments

Syntax:

constraint ($\sigma_1(Y)$) Y .SegmentsOf_a($\sigma_{1a}(Y, a)$) compostoDa
($(\sigma_{2,1}(Y, X_1)) X_1$.SegmentsOf_b₁($\sigma_{2b,1}(Y, X_1, b_1)$), ...,
($\sigma_{2,n}(Y, X_n)$) X_n .SegmentsOf_b_n($\sigma_{2b,n}(Y, X_n, b_n)$))

OCL template:

ComposedOfConstraintMulti^{SEL}_{TR/TR}($Y, \sigma_1(Y), a, \sigma_{1a}(Y, a),$
 $X_1, \sigma_{2,1}(Y, X_1), b_1, \sigma_{2b,1}(Y, X_1, b_1), \dots,$
 $X_n, \sigma_{2,n}(Y, X_n), b_n, \sigma_{2b,n}(Y, X_n, b_n)$) \equiv

context Y

inv: $\sigma_1(\text{self})$ implies

$\text{self.SegmentsOf_a}(\text{"}\sigma_{1a}(Y, a)\text{"}) \rightarrow$

$\text{forall}(t:\text{GU_Object} \mid t.\text{Equals}(\text{$

$X_1.\text{allInstances} \rightarrow$

$\text{select}(x_1:X_1 \mid \sigma_{2,1}(\text{self}, x_1)).$

$\text{SegmentsOf_b}_1(\text{"}\sigma_{2b,1}(Y, X_1, b_1)\text{"}) \rightarrow$

$\text{union}(\dots) \rightarrow$

...

$\text{union}(X_n.\text{allInstances} \rightarrow$

$\text{select}(x_n:X_n \mid \sigma_{2,n}(\text{self}, x_n)).$

$\text{SegmentsOf_b}_n(\text{"}\sigma_{2b,n}(Y, X_n, b_n)\text{"}) \rightarrow$

$\text{select}(a:\text{GU_Object} \mid t.\text{Contains}(a) \text{ or } t.\text{Equals}(a)) \rightarrow$

$\text{iterate}(b:\text{GU_Object}, \text{acc}:\text{GU_Object} = \emptyset \mid$

$\text{acc.gUnion}(b))$

)

GeoUML Model

Geometric Model and OCL Constraints Templates

geometries/segments

Syntax:

constraint ($\sigma_1(Y)$)Y.f compostoDa
($(\sigma_{2,1}(Y, X_1)) X_1$.SegmentsOf_b1($\sigma_{2b,1}(Y, X_1, b_1)$), ...,
($\sigma_{2,n}(Y, X_n)$) X_n .SegmentsOf_bn($\sigma_{2b,n}(Y, X_n, b_n)$))

OCL template:

ComposedOfConstraintMulti^{SEL}_{TR/TR}(Y, $\sigma_1(Y)$, f,
X₁, $\sigma_{2,1}(Y, X_1)$, b₁, $\sigma_{2b,1}(Y, X_1, b_1)$, ...,
X_n, $\sigma_{2,n}(Y, X_n)$, b_n, $\sigma_{2b,n}(Y, X_n, b_n)$) \equiv

context Y

inv: $\sigma_1(\text{self})$ implies

```
self.f.Equals(  
  X1.allInstances->  
    select(x1:X1| $\sigma_{2,1}(\text{self}, X_1)$ ).  
      SegmentsOf_b1(" $\sigma_{2b,1}(Y, X_1, b_1)$ ")->  
    union(...)->  
  ...  
  union(Xn.allInstances->  
    select(xn:Xn| $\sigma_{2,n}(\text{self}, x_n)$ ).  
      SegmentsOf_bn(" $\sigma_{2b,n}(Y, X_1, b_n)$ ")->  
  select(a:GU_Object| self.f.Contains(a)  
    or self.f.Equals(a))->  
  iterate(b:GU_Object, acc: GU_Object =  $\emptyset$  |  
    acc.gUnion(b))  
)
```

segments/geometries

Syntax:

constraint $(\sigma_1(Y))Y.$ SegmentsOf_a $(\sigma_{1a}(Y,a))$ compostoDa
 $((\sigma_{2,1}(Y,X_1))X_1.g_1, \dots, (\sigma_{2,n}(Y,X_n))X_n.g_n$

OCL template:

ComposedOfConstraintMulti^{SEL}_{TR/TR}(Y, $\sigma_1(Y)$, a, $\sigma_{1a}(Y,a)$,
 $X_1, \sigma_{2,1}(Y,X_1), g_1, \dots, X_n, \sigma_{2,n}(Y,X_n), g_n$) \equiv

context Y

inv: $\sigma_1(\text{self})$ implies

```
self.SegmentsOf_a(" $\sigma_{1a}(Y,a)$ ")->
  forall(t:GU_Object | t.Equals(
    X1.allInstances->select(x1:X1 |  $\sigma_{2,1}(\text{self}, x_1)$ )).g1->
    union(...)->
    ...
    union(Xn.allInstances->
      select(xn:Xn |  $\sigma_{2,n}(\text{self}, x_n)$ )).gn)->
  select(a:GU_Object | t.Contains(a) or t.Equals(a))->
  iterate(b:GU_Object, acc: GU_Object =  $\emptyset$  |
    acc.gUnion(b))
)
```

A.8.4 Variant on boundary and planar projection

Variant on boundary and planar projection

Definition of symbols:

Given a class Y with a geometric attribute f and a set of classes X_1, \dots, X_n with a geometric attribute g_1, \dots, g_n , the composition constraint from Y to X_1, \dots, X_n , variant on boundary or on planar projection, is defined as follows:

Syntax:

constraint $Y.f.BND$ compostoDa ($X_1.g_1, \dots, X_n.g_n$)

OCL template:

ComposedOfConstraintMulti^{B-}($Y, f, X_1, g_1, \dots, X_n, g_n$) \equiv

context Y

inv: $self.f.boundary().Equals(X_1.allInstances.g_1 \rightarrow$
 $union(X_2.allInstances.g_2) \rightarrow$

...

$union(X_n.allInstances.g_n) \rightarrow$

$select(a:GU_Object |$

$self.f.boundary().Contains(a) \text{ or}$

$self.f.boundary().Equals(a)) \rightarrow$

$iterate(b:GU_Object, acc: GU_Object = \emptyset |$
 $acc.gUnion(b))$

)

Syntax:

constraint $Y.f$ compostoDa ($X_1.g_1.PLN, \dots, X_n.g_n.PLN$)

OCL template:

ComposedOfConstraintMulti^{P-}($Y, f, X_1, g_1, \dots, X_n, g_n$) \equiv

context Y

inv: $self.f.Equals(X_1.allInstances.g_1.planar() \rightarrow$
 $union(X_2.allInstances.g_2.planar()) \rightarrow$

...

$union(X_n.allInstances.g_n.planar()) \rightarrow$

$select(a:GU_Object | self.f.Contains(a)$

$\text{ or } self.f.Equals(a)) \rightarrow$

$iterate(b:GU_Object, acc: GU_Object = \emptyset |$
 $acc.gUnion(b))$